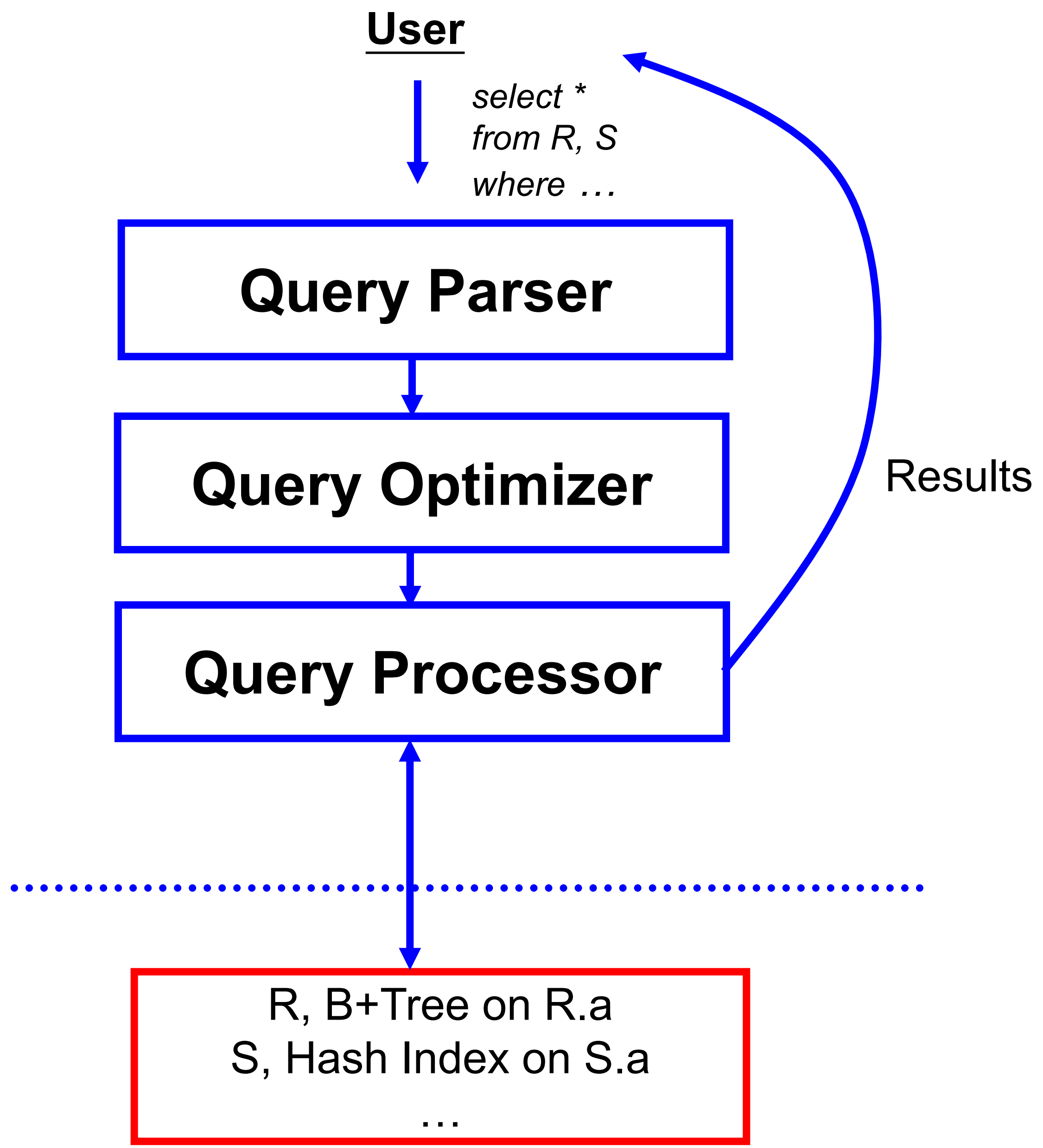
Query Processing

Overview

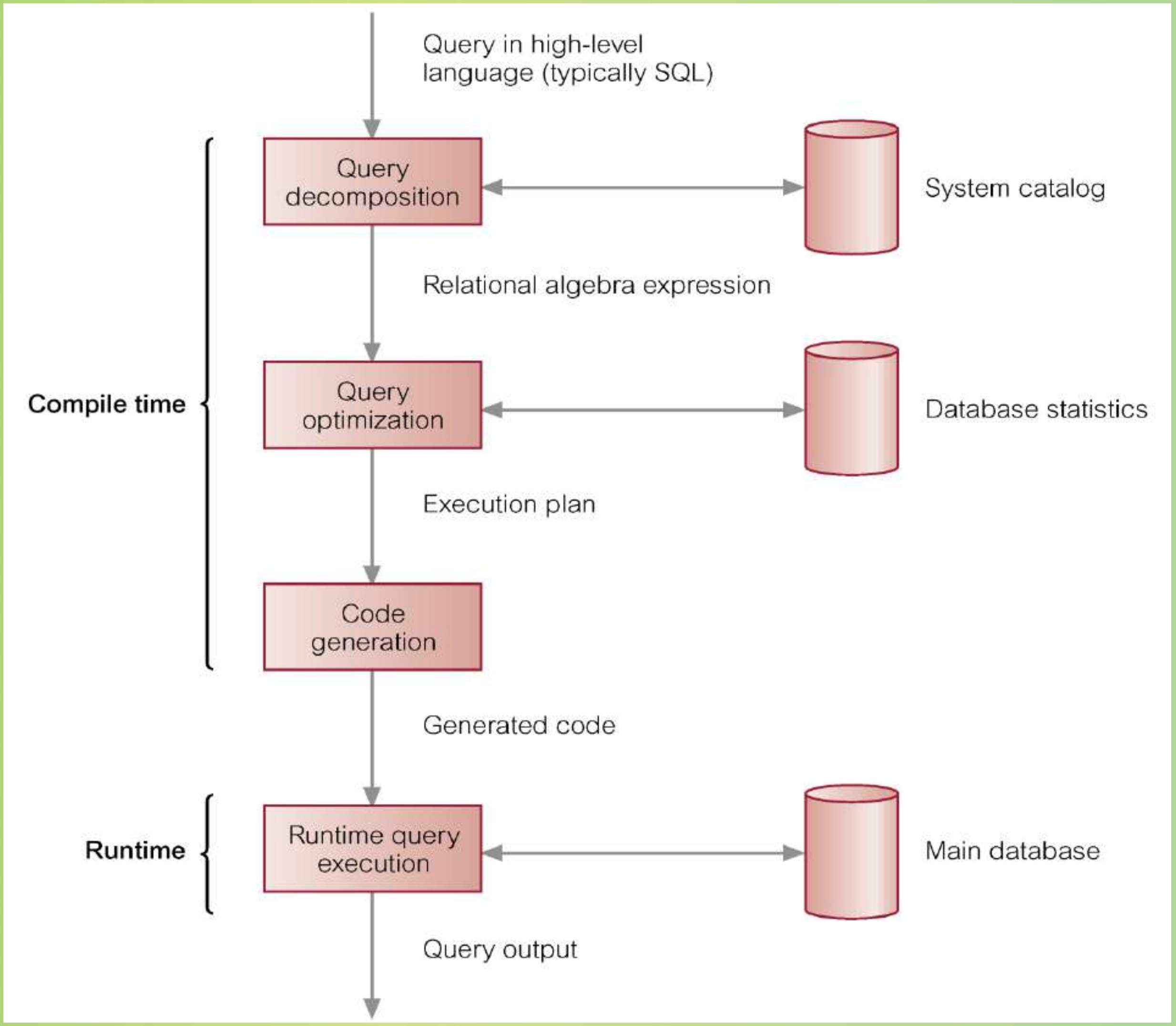


Resolve the references,
Syntax errors etc.
Converts the query to an internal format
relational algebra like

Find the *best* way to evaluate the query
- Which index to use ?
- What join method to use ?
- …

Read the data from the files
Do the query processing
- *joins, selections, aggregates*
- …

Phases of Query Processing



Dynamic versus Static Optimization

- Two times when first three phases of QP can be carried out:
  - dynamically every time query is run;
  - statically when query is first submitted.
- Advantages of dynamic QO arise from fact that information is up to date.
- Disadvantages are that performance of query is affected, time may limit finding optimum strategy.

Dynamic versus Static Optimization

- Advantages of static QO are removal of runtime overhead, and more time to find optimum strategy.
- Disadvantages arise from fact that chosen execution strategy may no longer be optimal when query is run.
- Could use a hybrid approach to overcome this.

“Cost”

- Complicated to compute
- We will focus on disk:
  - Number of I/Os ?
    - Not sufficient
    - Number of seeks matters a lot… why ?
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for *b* block transfers plus *S* seeks

    $$b * t_T + S * t_S$$
  - Measured in *seconds*

Selection Operation

- select * from person where SSN = "123"
- Option 1: <u>Sequential Scan</u>
  - Read the relation start to end and look for "123"
    - Can always be used (not true for the other options)
  - Cost
    - *Let b_r = Number of relation blocks*
    - Then:
      - 1 seek and br block transfers
    - So:
      - $t_S + b_r * t_T$ *sec*
    - *Improvements:*
      - *If SSN is a key, then can stop when found*
        - *So on average, $b_r/2$ blocks accessed*

Selection Operation

- select * from person where SSN = “123”
- Option 2 : <u>Binary Search:</u>
  - Pre-condition:
    - *The relation is sorted on SSN*
    - *Selection condition is an equality*
      - E.g. can’t apply to “*Name like ‘%424%’*”
  - Do binary search
    - Cost of finding the *first* tuple that matches
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
      - All I/Os are random, so need a seek for all

Selection Operation

- select * from person where SSN = “123”
- Option 3 : <u>Use Index</u>
  - Pre-condition:
    - *An appropriate index must exist*
  - Use the index
    - Find the first leaf page that contains the search key
    - Retrieve all the tuples that match by following the pointers
      - If primary index, the relation is sorted by the search key
        - Go to the relation and read blocks sequentially
      - If secondary index, must follow all points using the index

Selection Operation

- Selections involving ranges
  - *select * from accounts where balance > 100000*
  - *select * from matches where matchdate between '10/20/06' and '10/30/06'*
  - Option 1: Sequential scan
  - Option 2: Using an appropriate index
    - Can't use hash indexes for this purpose
    - Cost formulas:
      - Range queries == "equality" on "non-key" attributes

Selection Operation

- Complex selections
  - <u>Conjunctive</u>: *select * from accounts where balance > 100000 and SSN = "123"*
  - <u>Disjunctive</u>: *select * from accounts where balance > 100000 or SSN = "123"*
  - Option 1: **Sequential scan**
  - *(Conjunctive only)* Option 2: **Using an appropriate index *on one of the conditions***
    - E.g. Use SSN index to evaluate SSN = "123". Apply the second condition to the tuples that match
    - Or do the other way around (if index on balance exists)
    - Which is better ?
  - *(Conjunctive only)* Option 3: **Choose a multi-key index**
    - Not commonly available

Selection Operation

- Complex selections
  - <u>*Conjunctive*</u>*: select * from accounts where balance > 100000 and SSN = "123"*
  - <u>Disjunctive</u>: *select * from accounts where balance > 100000 or SSN = "123"*

  - **Option 4: Conjunction or disjunction of *record identifiers***
    - Use indexes to find all RIDs that match each of the conditions
    - Do an intersection (for conjunction) or a union (for disjunction)
    - Sort the records and fetch them in one shot
    - Called "Index-ANDing" or "Index-ORing"
  - Heavily used in commercial systems

Example 23.1 - Different Strategies

Find all Managers who work at a London branch.

```
SELECT *
FROM Staff s, Branch b
WHERE s.branchNo = b.branchNo AND
(s.position = ‘Manager’ AND b.city = ‘London’);
```

Example 23.1 - Different Strategies

- **Three equivalent RA queries are:**

(1) $\sigma_{(position='Manager') \wedge (city='London') \wedge (Staff.branchNo=Branch.branchNo)}$ **(Staff X Branch)**

(2) $\sigma_{(position='Manager') \wedge (city='London')}$**(Staff** $\bowtie_{Staff.branchNo=Branch.branchNo}$ **Branch)**

(3) **(**$\sigma_{position='Manager'}$**(Staff))** $\bowtie_{Staff.branchNo=Branch.branchNo}$ **(**$\sigma_{city='London'}$ **(Branch))**

Example 23.1 - Different Strategies

- **Assume:**
  - **1000 tuples in Staff; 50 tuples in Branch;**
  - **50 Managers; 5 London branches;**
  - **no indexes or sort keys;**
  - **results of any intermediate operations stored on disk;**
  - **cost of the final write is ignored;**
  - **tuples are accessed one at a time.**

Example 23.1 - Cost Comparison

- Cost (in disk accesses) are:

(1) (1000 + 50) + 2*(1000 * 50) = 101 050

(2) 2*1000 + (1000 + 50) = 3 050

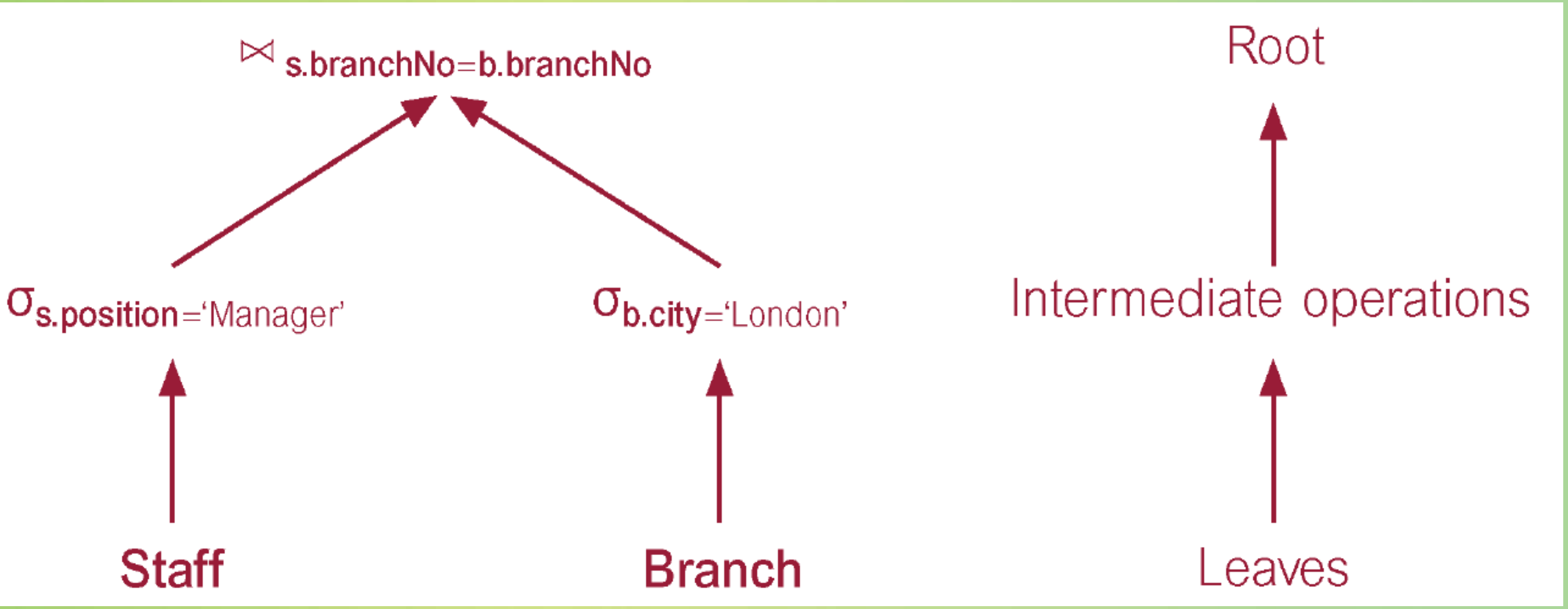
(3) 1000 + 2*50 + 5 + (50 + 5) = 1 160

- Cartesian product and join operations much more expensive than selection, and third option significantly reduces size of relations being joined together.

Analysis

- Finally, query transformed into some internal representation more suitable for processing.
- Some kind of query tree is typically chosen, constructed as follows:
  - Leaf node created for each base relation.
  - Non-leaf node created for each intermediate relation produced by RA operation.
  - Root of tree represents query result.
  - Sequence is directed from leaves to root.

Example 23.1 - R.A.T.



Query Processing

- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together…

Join

- *select * from R, S where R.a = S.a*
  - R called *outer relation*
  - S called *inner relation*
  - Called an "*equi-join*"
- *select * from R, S where |R.a – S.a | < 0.5*
  - *Not an "equi-join"*

- Option 1: Nested-loops

  for each tuple r in R

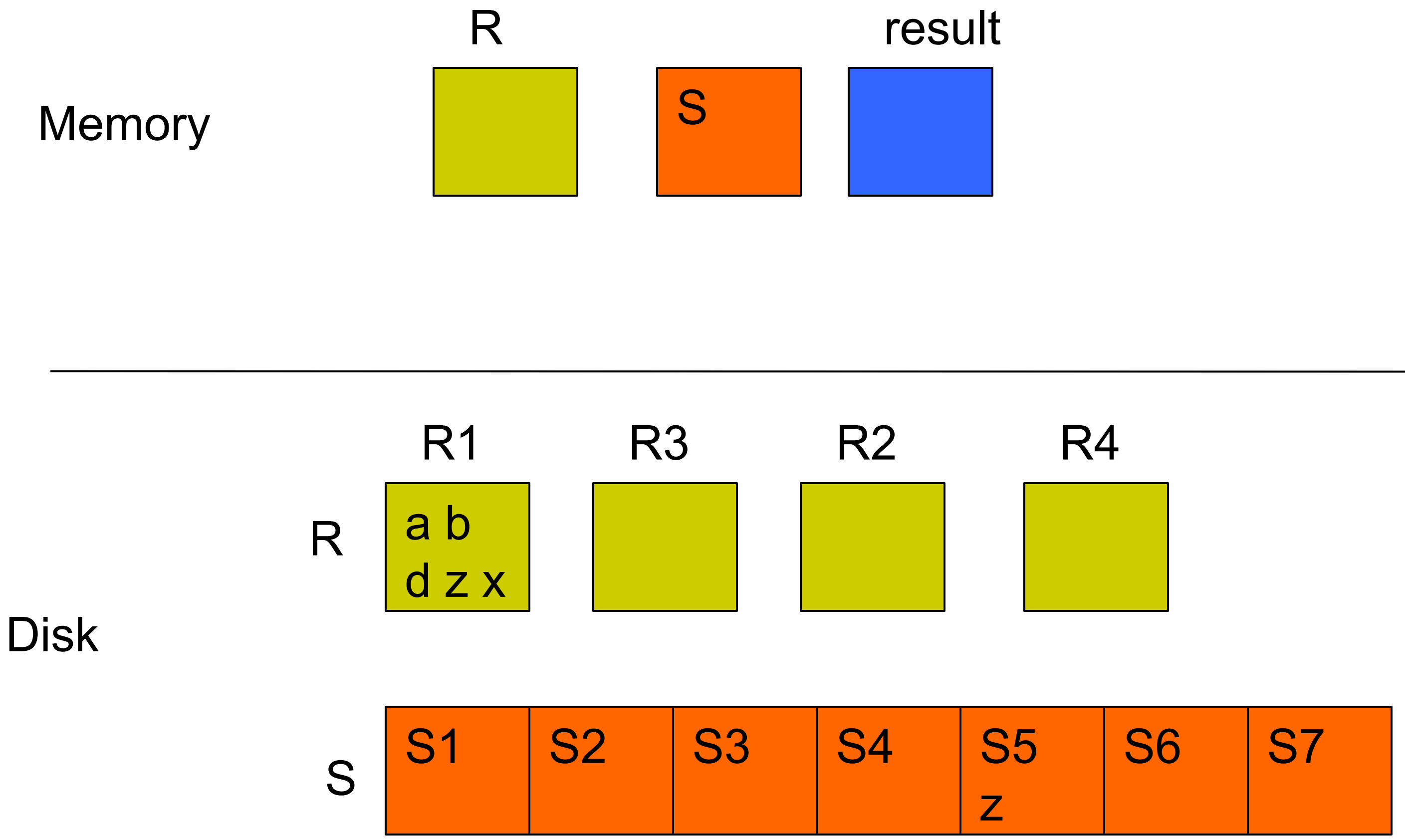
      for each tuple s in S

          check if r.a = s.a (or whether |r.a – s.a| < 0.5)
- Can be used for any join condition

Nested-loops Join

- Cost ? Depends on the actual values of parameters, especially memory
- b_r, b_s → *Number of blocks of R and S*
- n_r, n_s → *Number of tuples of R and S*
- Case 1: Minimum memory required = 3 blocks
  - One to hold the current *R* block, one for current S block, one for the result being produced
  - Blocks transferred:
    - Must scan *R* tuples once: b_r
    - For each tuple in R, must scan *S:* $n_r * b_s$
  - Seeks
    - $n_r + b_r$

Example



Nested-loops Join

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $n_r * b_s + b_r$
  - Seeks: $n_r + b_r$

In this case, disk blocks in R is not ordered and data tuples in a disk block are not ordered.

- Example:
  - Number of records -- *R:* $n_r = 10{,}000$, *S:* $n_s = 5000$
  - Number of blocks -- *R:* $b_r = 400$, *S:* $b_s = 100$
- Then:
  - blocks transferred: 10000 * 100 + 400 = 1,000,400
  - seeks: 10400
- What if we were to switch R and S ?
  - 2,000,100 block transfers, 5100 seeks

Nested-loops Join

- <u>Case 2: *S* fits in memory</u>
  - Blocks transferred: $b_s + b_r$
  - Seeks: *2* (Assume that R is also sequential read into a block)
- Example:
  - Number of records -- R: $n_r = 10{,}000$, S: $n_s = 5000$
  - Number of blocks -- R: $b_r = 400$, S: $b_s = 100$
- Then:
  - blocks transferred: 400 + 100 = 500
  - seeks: 2
- This is orders of magnitude difference

Block Nested-loops Join

- Simple modification to "nested-loops join"
  - Block at a time

for each block B_r in R

for each block B_s in S

for each tuple r in Br

for each tuple s in Bs

check if r.a = s.a (or whether |r.a – s.a| < 0.5)

Block Nested-loops Join

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $b_r * b_s + b_r$
  - Seeks: $b_r + b_r$
- Case 2: S fits in memory
  - Blocks transferred: $b_s + b_r$
  - Seeks: *2* (Assume that R is also sequential read into a block)
- What about in between ?
  - There are 50 blocks, but *S* is 100 blocks

Block Nested-loops Join

- Case 3: 50 blocks (S = 100 blocks) ?

  for each block in R

  for each group of 48 blocks in S

  for each tuple r in one block

  for each tuple s in each group of 48 blocks in S

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- Why is this good ?
  - We only have to read *S* a total of $b_s/48$ *times (instead of* b_s *times)*
  - Blocks transferred: $b_r * (b_s / 48) + b_r$
  - Seeks: 2 (if R and S are ordered files)

Index Nested-loops Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- Nested-loops

    for each tuple r in R

    for each tuple s in S

    check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- Suppose there is an index on *S.a*
- *Why not use the index instead of the inner loop ?*

    for each tuple r in R

    use the index to find S tuples with S.a = r.a

Index Nested-loops Join

- Cost of the join:
  - $b_r (t_T + t_S) + n_r * c$
  - *c == the cost of index access*
    - *Computed using the formulas discussed earlier*

Index Nested-loops Join

- Restricted applicability
  - An appropriate index must exist
  - What about |*R.a* – *S.a*| < *5* ?
- Great for queries with joins and selections

*select **

from accounts, customers

where accounts.customer-SSN = customers.customer-SSN and

accounts.acct-number = “A-101”

- Only need to access one SSN from the other relation

Notes

- Block Nested-loops join
  - Can always be applied to various join conditions
  - If the smaller relation fits in memory, then cost:
    - $b_r + b_s$
    - This is the best we can hope if we have to read the relations once each
  - CPU cost of the inner loop is high
  - Typically used when the smaller relation is really small (few tuples) and index nested-loops can't be used
- Index Nested-loops join
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - select balance from customer, accounts where customer.name = "j. s." and customer.SSN = accounts.SSN

Hash Join

- Case 1: Smaller relation *(S)* fits in memory
- Nested-loops join:

  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a

- Cost: $b_r + b_s$ transfers, 2 seeks (R and S are sorted)
- The inner loop is not exactly cheap (high CPU cost)

- Hash join:

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a

Hash Join

- Case 1: Smaller relation *(S)* fits in memory
- Hash join:

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a
- Cost: $b_r + b_s$ transfers, 2 seeks (unchanged)
- Why good ?
  - CPU cost is much better (even though we don't care about it too much)
  - Performs much better than nested-loops join when *S* doesn't fit in memory

Hash Join

- <u>Case 2: Smaller relation *(S)* doesn't fit in memory</u>
- Two "phases"
- Phase 1:
  - Read the relation *R* block by block and partition it using a hash function, *h1(a)*
    - Create one partition for each possible value of *h1(a)*
  - Write the partitions to disk
    - R gets partitioned into R1, R2, …, Rk
  - Similarly, read and partition S, and write partitions S1, S2, …, Sk to disk
  - Only requirement:
    - Each S partition fits in memory

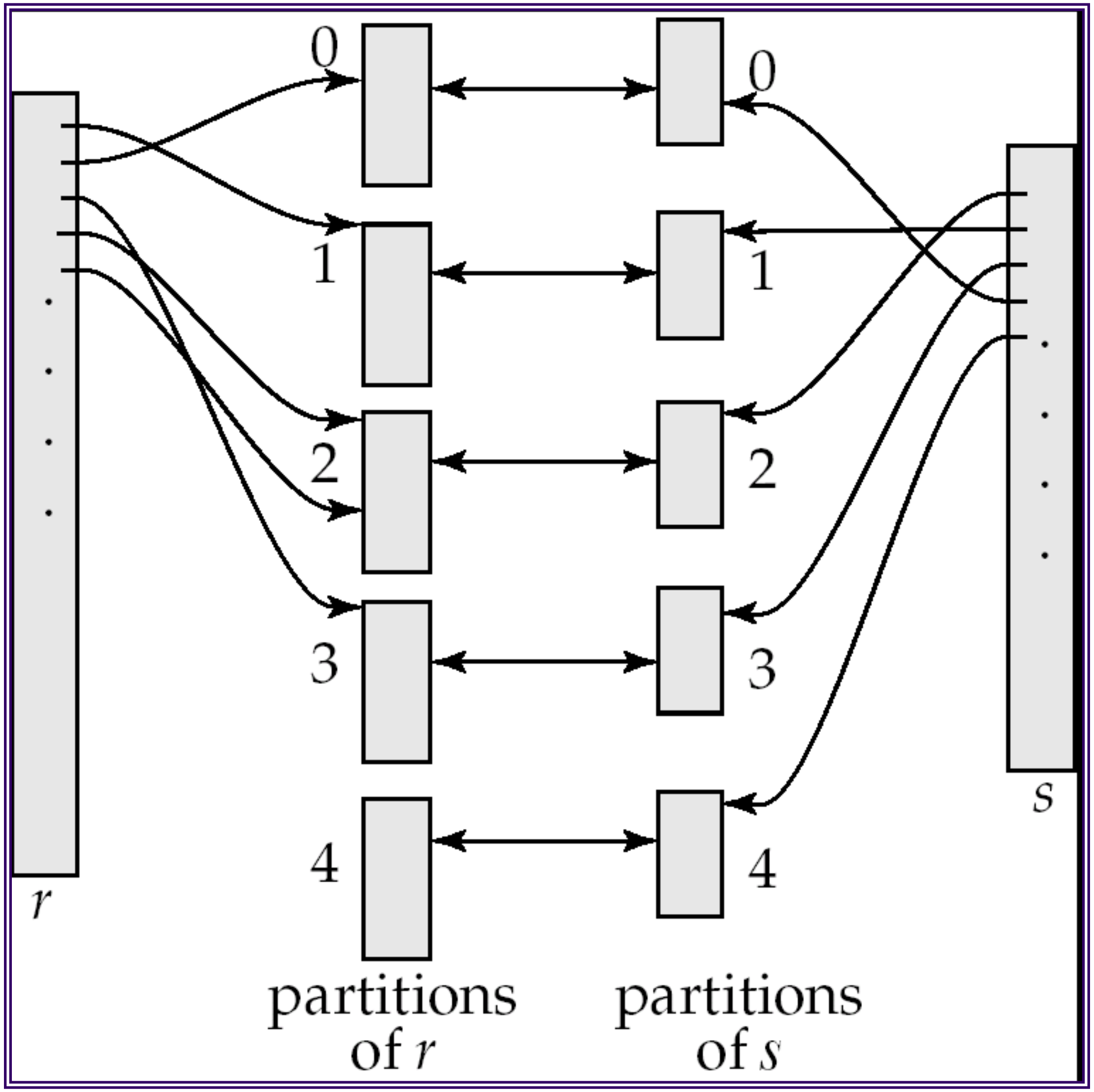
Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 2:
  - Read S1 into memory, and bulid a hash index on it (S1 fits in memory)
    - Using a different hash function, $h_2(a)$
  - Read R1 block by block, and use the hash index to find matches.
  - Repeat for S2, R2, and so on.

Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases":
- Phase 1:
  - Partition the relations using one hash function, $h_1(a)$
- Phase 2:
  - Read S_i into memory, and build a hash index on it (S_i fits in memory)
  - Read R_i, and use the hash index to find matches.
- Cost
  - $3(b_r + b_s) + 4 * n_h$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks (Read 13.5.5.4 in the reference text book)
    - Where b_b is the size of each output buffer
  - Much better than Nested-loops join under the same conditions

Hash Join



Hash Join: Issues

- How to guarantee that the partitions of *S* all fit in memory ?
  - S = 10000 blocks, Memory = M = 100 blocks
  - Use a hash function that hashes to 100 different values ?
    - Eg. $h1(a) = a \% 100$?
  - Problem: Impossible to guarantee uniform split
    - Some partitions will be larger than 100 blocks, some will be smaller
  - Use a hash function that hashes to $100*f$ different values
    - f is called fudge factor, typically around 1.2
    - So we may consider $h1(a) = a \% 120$.
    - This is okay IF a is uniformly distributed

Hash Join: Issues

- Memory required ?
  - R = 10000 blocks, Memory = M = 100 blocks
  - 120 different partitions
  - During phase 1:
    - Need 1 block for storing *R (block by block)*
    - Need 120 blocks for storing each partition of *R*
  - At least 121 blocks of memory
  - We only have 100 blocks
- Typically need *SQRT(|R| * f)* blocks of memory
- If R is 10000 blocks, and f = 1.2, need 110 blocks of memory
- If memory = 10000 blocks = 10000 * 4 KB = 40MB
  - Then, R can be as large as *10000*10000/1.2* blocks = *333 GB*

Hash Join: Issues

- What if we don't have enough memory ?
  - *Recursive Partitioning*
  - Rarely used, but can be done

- What if the hash function turns out to be bad ?
  - We used *h1(a) = a % 100*
  - Turns out all *values of a* are multiple of 100
  - So *h1(a) is always = 0*
- Called *hash-table overflow*
- Overflow avoidance: Use a good hash function
- Overflow resolution: Repartition using a different hash function

Hybrid Hash Join

- Motivation:
  - R = 10000 blocks, S = 101 blocks, M = 100 blocks
  - *S* doesn't fit in memory
- Approach: Use a hash function such that *S1 = 90 blocks, and S2 = 10 blocks (Try to keep more data records of S in memory)* *(Why S ?)*
- Steps:
  - Read S1, and partition it
    - Write S2 to disk
    - Keep S1 in memory, and build a hash table on it
  - Read R1, and partition it
    - Write R2 to disk
    - Probe using R1 directly into the hash table
  - Saves huge amounts of I/O

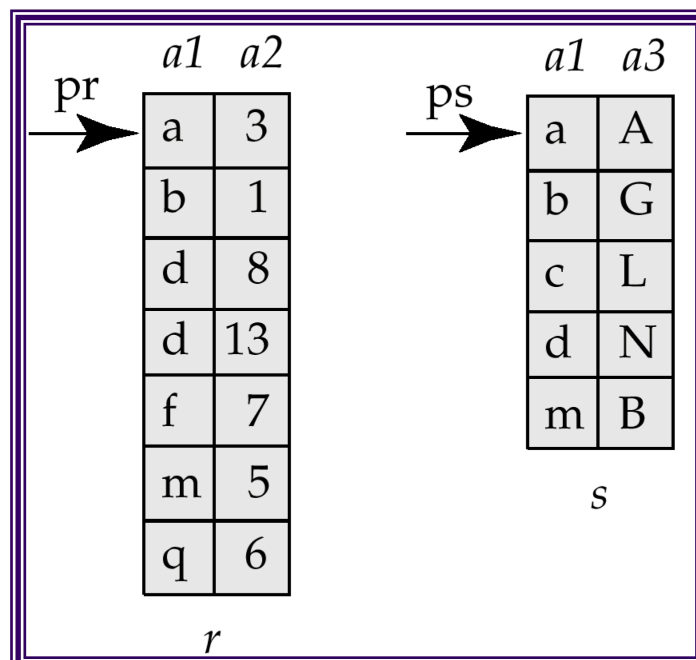
So far

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - select balance from customer, accounts where customer.name = “j. s.” and customer.SSN = accounts.SSN
- Hash joins
  - Join algorithm of choice when the relations are large
  - Only applies to equi-joins (since it is hash-based)
- Hybrid hash join
  - An optimization on hash join that is always implemented

Merge-Join (Sort-merge join)

- Pre-condition:
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this algorithms
- Called “sort-merge join” sometimes

*select **
from r, s
where r.a1 = s.a1



Step:

1. Compare the tuples at pr and ps

2. Move pointers down the list

- Depending on the join condition

3. Repeat

Merge-Join (Sort-merge join)

- Cost:
  - If the relations sorted, then just
    - $b_r + b_s$ block transfers, some seeks depending on memory size
  - What if not sorted ?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on *a1*
  - This might make further operations easier to do
    - E.g. duplicate elimination

Group By and Aggregation

select a, count(b)
from R
group by a;

- Hash-based algorithm
- Steps:
  - Create a hash table on *a,* and keep the *count(b) so far*
  - Read *R* tuples one by one
  - For a new *R tuple, "r"*
    - Check if *r.a* exists in the hash table
    - If yes, increment the count
    - If not, insert a new value

Group By and Aggregation

select a, count(b)
from R
group by a;

- Sort-based algorithm
- Steps:
  - Sort *R* on *a*
  - Now all tuples in a single group are continuous
  - Read tuples of *R (sorted)* one by one and compute the aggregates

Duplicate Elimination

select distinct a
from R ;

- Best done using sorting – Can also be done using hashing
- Steps:
  - Sort the relation *R*
  - Read tuples of *R* in sorted order
  - prev = null;
  - for each tuple *r* in *R (sorted)*
    - if *r != prev* then
      - Output *r*
      - *prev* = *r*
    - else
      - Skip *r*

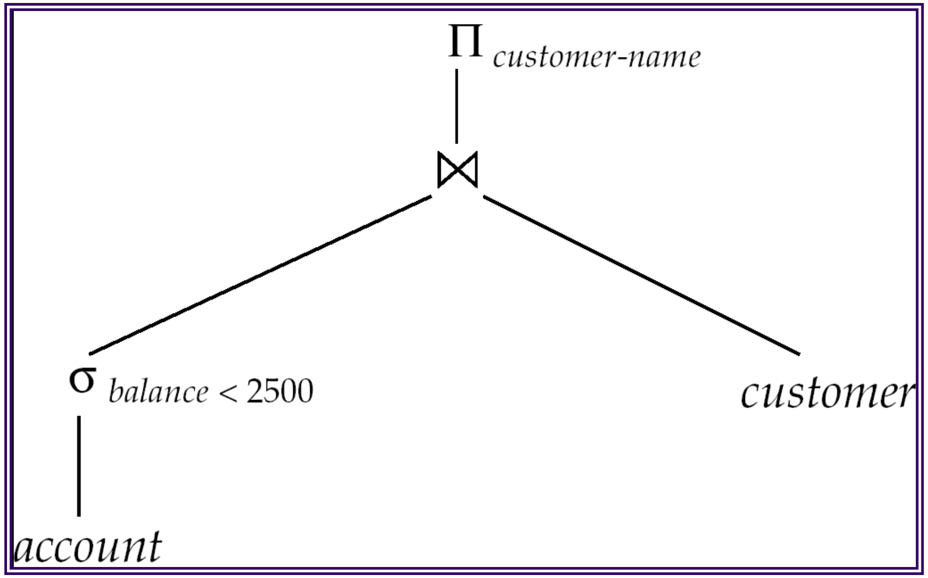
Set operations

*(select * from R) union (select * from S) ;*
*(select * from R) intersect (select * from S) ;*
*(select * from R) union all (select * from S) ;*
*(select * from R) intersect all (select * from S) ;*

- Remember the rules about duplicates
- "union all": just append the tuples of *R and S*
- *"*union": append the tuples of *R and S, and do duplicate elimination*
- *"intersection"*: similar to joins
  - Find tuples of R and S that are identical on all attributes
  - Can use *hash-based or sort-based algorithm*

Evaluation of Expressions

select customer-name
from account a, customer c
where a.SSN = c.SSN and
a.balance < 2500



- Two options:
  - Materialization
  - Pipelining

Evaluation of Expressions

- Materialization
  - Evaluate each expression separately
    - Store its result on disk in *temporary relations*
    - Read it for next operation
- Pipelining
  - Evaluate multiple operators simultaneously
  - Skip the step of going to disk
  - Usually faster, but requires more memory
  - Also not always possible..
    - E.g. Sort-Merge Join

Pipelining

- Iterator Interface
- Each operator implements:
  - *init(): Initialize the state*
  - *get_next(): get the next tuple from the operator*
  - *close(): Finish and clean up*
- Sequential Scan:
  - *init():* open the file
  - *get_next():* get the next tuple from file
  - *close():* close the file
- Execute by repeatedly calling *get_next()* at the root

Transformation Rules for RA Operations

Conjunctive Selection operations can cascade into individual Selection operations (and vice versa).

$$\sigma_{p \wedge q \wedge r}(R) = \sigma_p(\sigma_q(\sigma_r(R)))$$

- **Sometimes referred to as cascade of Selection.**

$$\sigma_{branchNo='B003' \wedge salary>15000}(Staff) = \sigma_{branchNo='B003'}(\sigma_{salary>15000}(Staff))$$

Transformation Rules for RA Operations

Commutativity of Selection.

$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$

- **For example:**

$\sigma_{branchNo='B003'}(\sigma_{salary>15000}(Staff)) =$
$\sigma_{salary>15000}(\sigma_{branchNo='B003'}(Staff))$

Transformation Rules for RA Operations

In a sequence of Projection operations, only the last in the sequence is required.

$$\Pi_L \Pi_M \ldots \Pi_N(R) = \Pi_L(R)$$

- For example:

$$\Pi_{lName} \Pi_{branchNo,\ lName}(Staff) = \Pi_{lName}(Staff)$$

Transformation Rules for RA Operations

Commutativity of Selection and Projection.

- **If predicate p involves only attributes in projection list, Selection and Projection operations commute:**

$$\Pi_{Ai, \ldots, Am}(\sigma_p(R)) = \sigma_p(\Pi_{Ai, \ldots, Am}(R))$$

where $p \in \{A_1, A_2, \ldots, A_m\}$

- **For example:**

$$\Pi_{fName, lName}(\sigma_{lName='Beech'}(Staff)) =$$
$$\sigma_{lName='Beech'}(\Pi_{fName,lName}(Staff))$$

Transformation Rules for RA Operations

Commutativity of Theta join (and Cartesian product).

$$R \bowtie_p S = S \bowtie_p R$$

$$R \times S = S \times R$$

Rule also applies to Equijoin and Natural join. For example:

$$\text{Staff} \bowtie_{\text{staff.branchNo=branch.branchNo}} \text{Branch} =$$

$$\text{Branch} \bowtie_{\text{staff.branchNo=branch.branchNo}} \text{Staff}$$

Transformation Rules for RA Operations

Commutativity of Selection and Theta join (or Cartesian product).

- **If selection predicate involves only attributes of one of join relations, Selection and Join (or Cartesian product) operations commute:**

$$\sigma_p(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r S$$

$$\sigma_p(R \times S) = (\sigma_p(R)) \times S$$

$$\text{where } p \in \{A_1, A_2, \ldots, A_n\}$$

Transformation Rules for RA Operations

- **If selection predicate is conjunctive predicate having form ($p \wedge q$), where p only involves attributes of R, and q only attributes of S, Selection and Theta join operations commute as:**

$$\sigma_{p \wedge q}(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r (\sigma_q(S))$$

$$\sigma_{p \wedge q}(R \times S) = (\sigma_p(R)) \times (\sigma_q(S))$$

Transformation Rules for RA Operations

- **For example:**

$$\sigma_{position='Manager' \wedge city='London'}(Staff \bowtie_{Staff.branchNo=Branch.branchNo} Branch) =$$

$$(\sigma_{position='Manager'}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\sigma_{city='London'}(Branch))$$

Transformation Rules for RA Operations

Commutativity of Projection and Theta join (or Cartesian product).

- **If projection list is of form $L = L_1 \cup L_2$, where L_1 only has attributes of R, and L_2 only has attributes of S, provided join condition only contains attributes of L, Projection and Theta join commute:**

$$\Pi_{L1 \cup L2}(R \bowtie_r S) = (\Pi_{L1}(R)) \bowtie_r (\Pi_{L2}(S))$$

Transformation Rules for RA Operations

- If join condition contains additional attributes not in L ($M = M_1 \cup M_2$ where M_1 only has attributes of R, and M_2 only has attributes of S), a final projection operation is required:

$$\Pi_{L1 \cup L2}(R \bowtie_r S) = \Pi_{L1 \cup L2}((\Pi_{L1 \cup M1}(R)) \bowtie_r (\Pi_{L2 \cup M2}(S)))$$

Transformation Rules for RA Operations

- **For example:**

$$\Pi_{position,city,branchNo}(Staff \bowtie_{Staff.branchNo=Branch.branchNo} Branch) =$$
$$(\Pi_{position,\ branchNo}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\Pi_{city,\ branchNo}(Branch))$$

- **and using the latter rule:**

$$\Pi_{position,\ city}(Staff \bowtie_{Staff.branchNo=Branch.branchNo} Branch) =$$
$$\Pi_{position,\ city}((\Pi_{position,\ branchNo}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\Pi_{city,\ branchNo}(Branch)))$$

Transformation Rules for RA Operations

Commutativity of Union and Intersection (but not set difference).

$R \cup S = S \cup R$

$R \cap S = S \cap R$

Transformation Rules for RA Operations

Commutativity of Selection and set operations (Union, Intersection, and Set difference).

$$\sigma_p(R \cup S) = \sigma_p(S) \cup \sigma_p(R)$$

$$\sigma_p(R \cap S) = \sigma_p(S) \cap \sigma_p(R)$$

$$\sigma_p(R - S) = \sigma_p(S) - \sigma_p(R)$$

Transformation Rules for RA Operations

Commutativity of Projection and Union.

$$\Pi_L(R \cup S) = \Pi_L(S) \cup \Pi_L(R)$$

Associativity of Union and Intersection (but not Set difference).

$$(R \cup S) \cup T = S \cup (R \cup T)$$

$$(R \cap S) \cap T = S \cap (R \cap T)$$

Transformation Rules for RA Operations

Associativity of Theta join (and Cartesian product).

- **Cartesian product and Natural join are always associative:**

  $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

  $(R \times S) \times T = R \times (S \times T)$

- **If join condition q involves attributes only from S and T, then Theta join is associative:**

  $(R \bowtie_{p} S) \bowtie_{q \wedge r} T = R \bowtie_{p \wedge r} (S \bowtie_{q} T)$

Transformation Rules for RA Operations

- **For example:**

$(\text{Staff} \bowtie_{\text{Staff.staffNo=PropertyForRent.staffNo}} \text{PropertyForRent}) \bowtie_{\text{ownerNo=Owner.ownerNo} \wedge \text{staff.lName=Owner.lName}} \text{Owner} =$

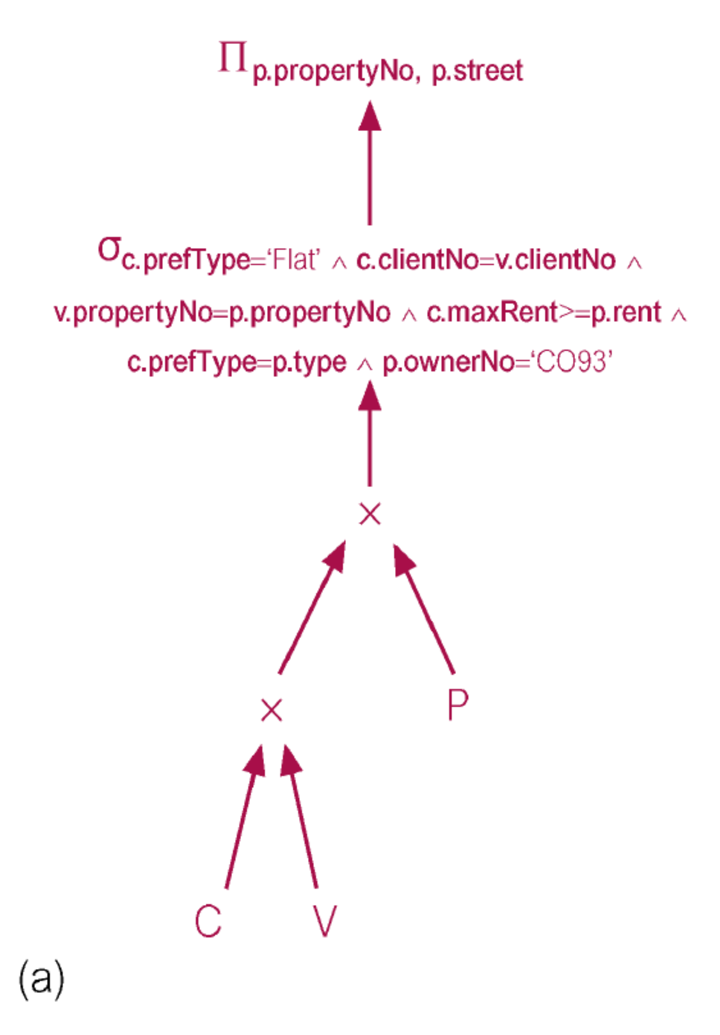
$\text{Staff} \bowtie_{\text{staff.staffNo=PropertyForRent.staffNo} \wedge \text{staff.lName=lName}} (\text{PropertyForRent} \bowtie_{\text{ownerNo}} \text{Owner})$

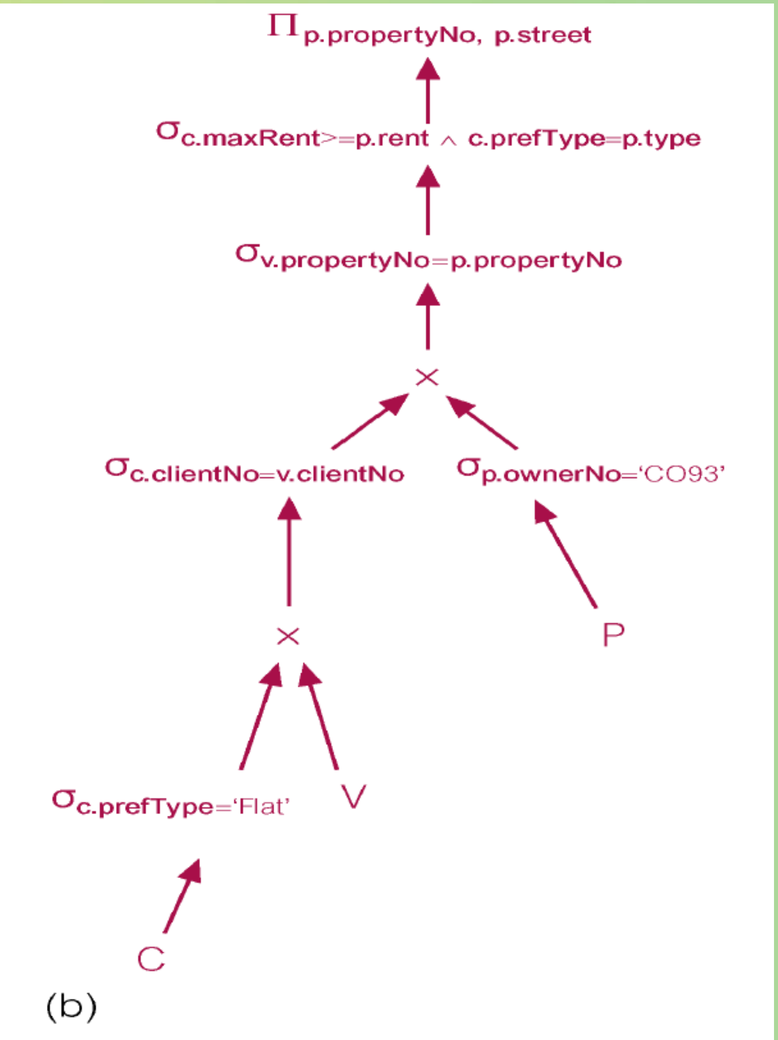
Example 23.3 Use of Transformation Rules

For prospective renters of flats, find properties that match requirements and owned by CO93.

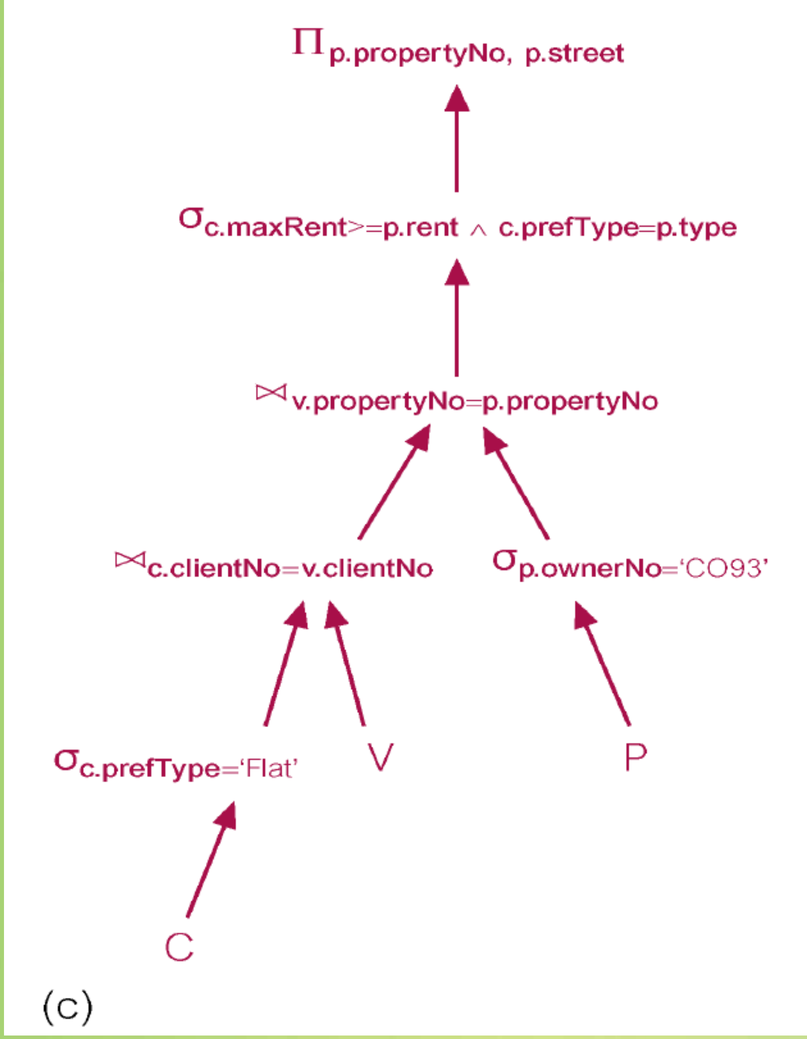
```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE   c.prefType = 'Flat' AND
        c.clientNo = v.clientNo AND
        v.propertyNo = p.propertyNo AND
        c.maxRent >= p.rent AND
        c.prefType = p.type AND
        p.ownerNo = 'CO93';
```

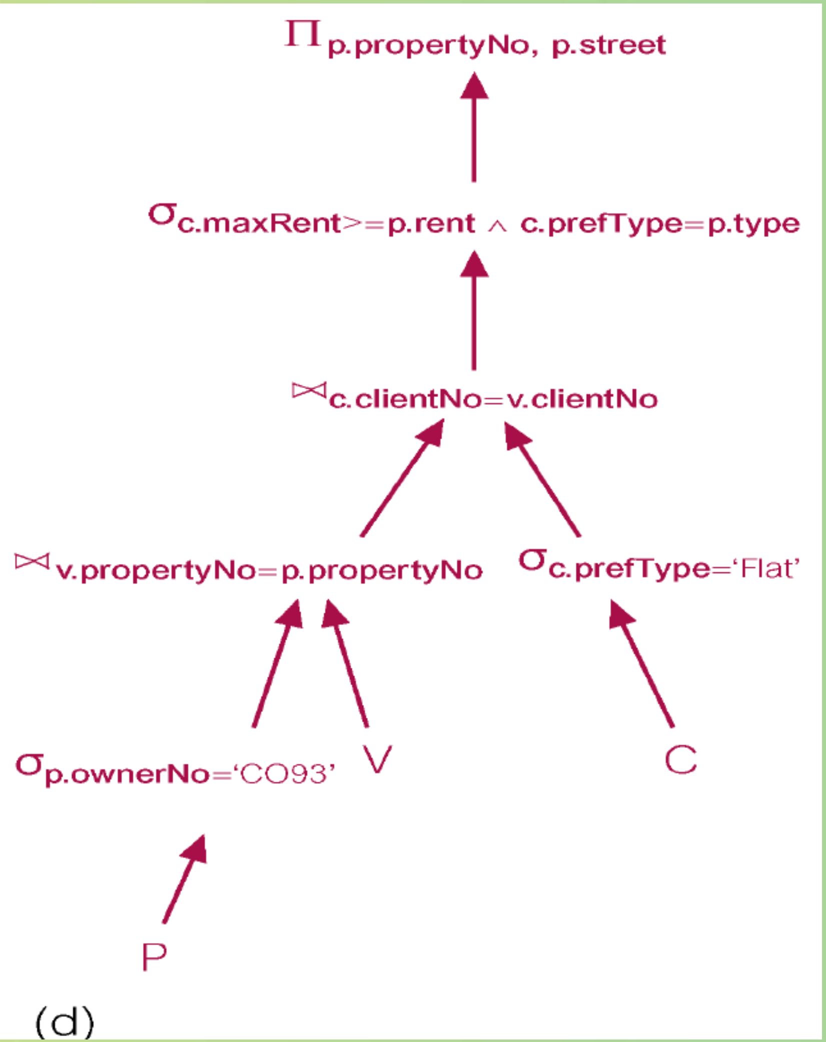
Example 23.3 Use of Transformation Rules



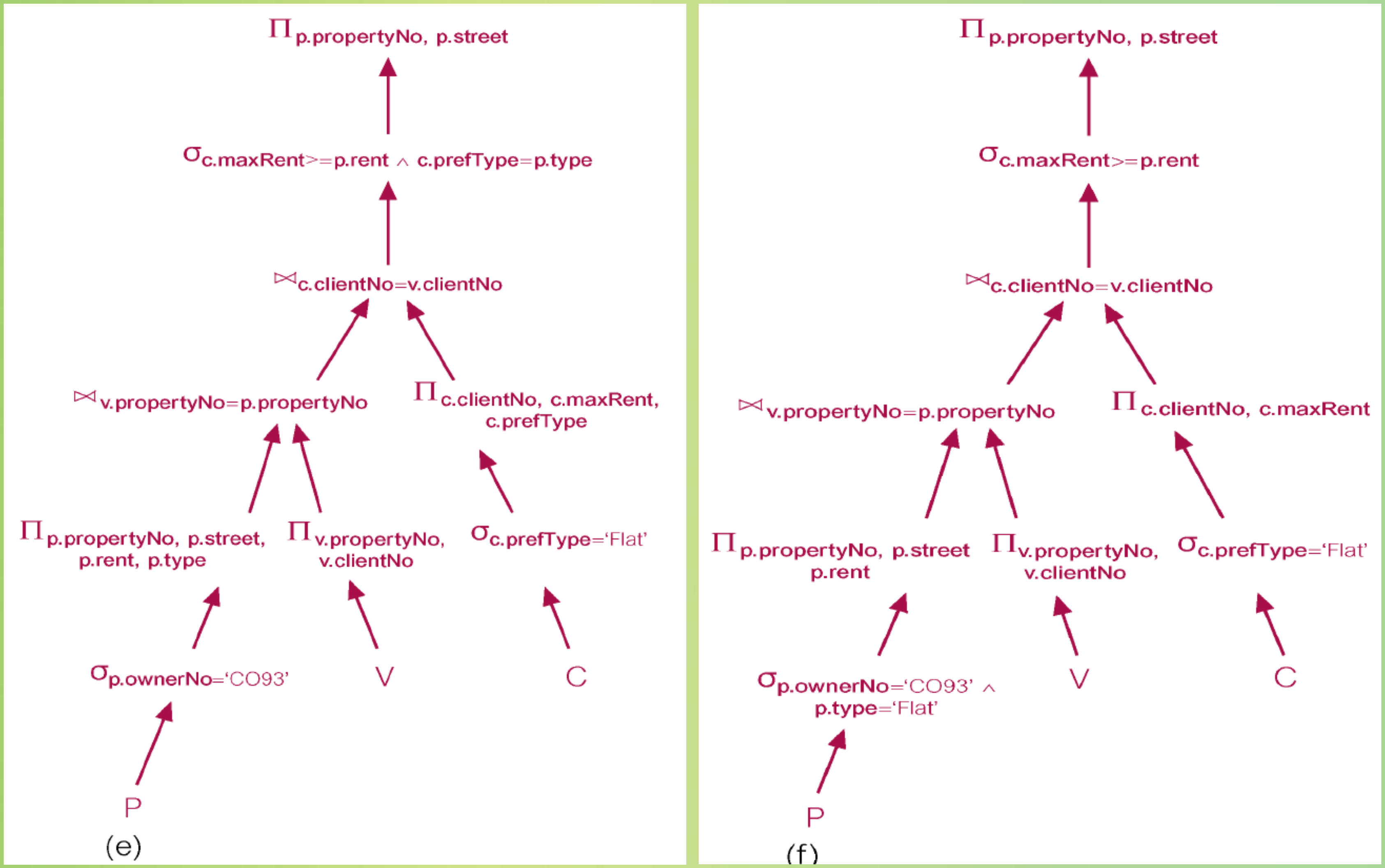


Example 23.3 Use of Transformation Rules





Example 23.3 Use of Transformation Rules



Heuristic Processing Strategies

- Perform Selection operations as early as possible.
  - Keep predicates on same relation together.
- Combine Cartesian product with subsequent Selection whose predicate represents join condition into a Join operation.
- Use associativity of binary operations to rearrange leaf nodes so leaf nodes with most restrictive Selection operations executed first.

Heuristical Processing Strategies

- **Perform Projection as early as possible.**
  - **Keep projection attributes on same relation together.**
- **Compute common expressions once.**
  - **If common expression appears more than once, and result not too large, store result and reuse it when required.**
  - **Useful when querying views, as same expression is used to construct view each time.**

Cost Estimation for RA Operations

- Many different ways of implementing RA operations.
- Aim of QO is to choose most efficient one.
- Use formulae that estimate costs for a number of options, and select one with lowest cost.
- Consider only cost of disk access, which is usually dominant cost in QP.
- Many estimates are based on cardinality of the relation, so need to be able to estimate this.

Database Statistics

- **Success of estimation depends on amount and currency of statistical information DBMS holds.**
- **Keeping statistics current can be problematic.**
- **If statistics updated every time tuple is changed, this would impact performance.**
- **DBMS could update statistics on a periodic basis, for example nightly, or whenever the system is idle.**

Typical Statistics for Relation R

nTuples(R) - number of tuples in R.

bFactor(R) - blocking factor of R.

nBlocks(R) - number of blocks required to store R:

nBlocks(R) = [nTuples(R)/bFactor(R)]

Typical Statistics for Attribute A of Relation R

$nDistinct_A(R)$ - number of distinct values that appear for attribute A in R.

$min_A(R)$, $max_A(R)$ - minimum and maximum possible values for attribute A in R.

$SC_A(R)$ - *selection cardinality* of attribute A in R.

> **Average number of tuples that satisfy an equality condition on attribute A.**

Statistics for Multilevel Index I on Attribute A

$nLevels_A(I)$ - number of levels in I.

$nLfBlocks_A(I)$ - number of leaf blocks in I.

References

- SQL query optimization http://redbook.cs.berkeley.edu/redbook3/lec7.html