Introduction to Database Management Systems

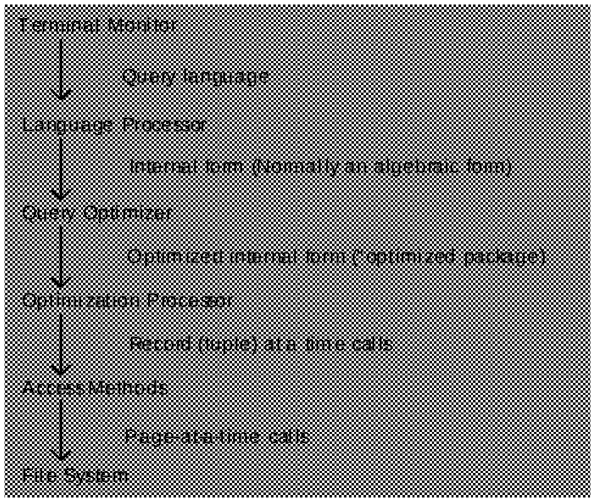
CS470

Query Processing and Optimization
Ch. 15

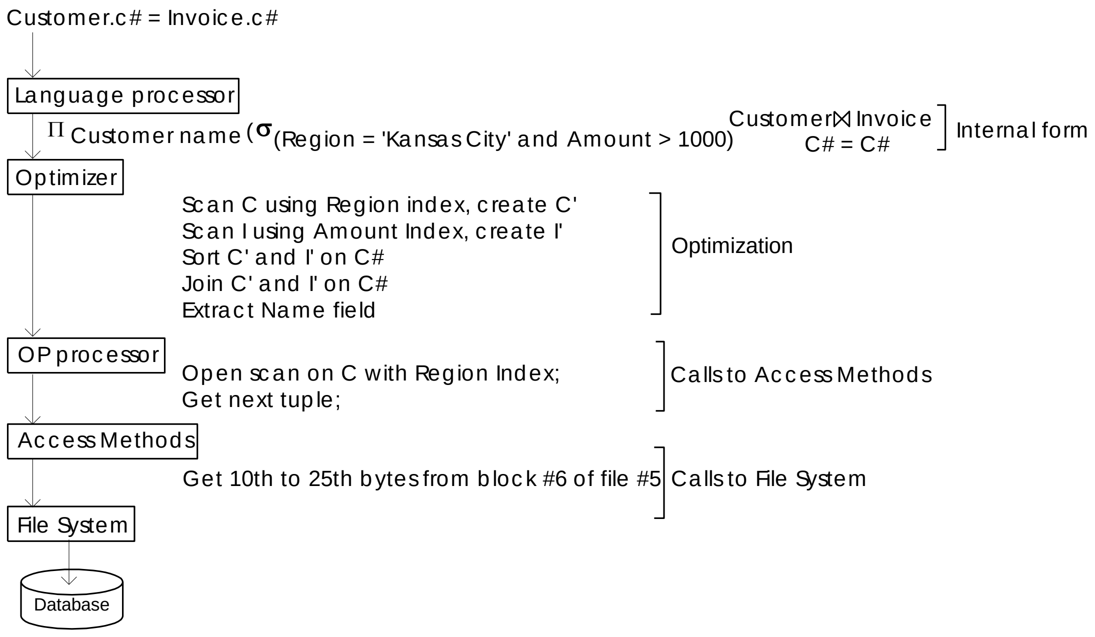
V Kumar
School of Computing and Engineering
University of Missouri-Kansas City

Query Processing and Optimization

Steps in query processing:



Example: **Select** Customer name
From Customer, Invoice
Where region = 'Kansas City and Amount > 1000


Customer.c# = Invoice.c#
Language processor
Π Customer name (σ (Region = 'Kansas City' and Amount > 1000) Customer ⋈ Invoice C# = C#
Internal form
Optimizer
Scan C using Region index, create C'
Scan I using Amount Index, create I'
Sort C' and I' on C#
Join C' and I' on C#
Extract Name field
Optimization
OP processor
Open scan on C with Region Index;
Get next tuple;
Calls to Access Methods
Access Methods
Get 10th to 25th bytes from block #6 of file #5
Calls to File System
File System
Database

Query optimization: There are many ways (access paths) for accessing desired file/record. The optimizer selects the most efficient (cheapest) access path for accessing the data.

Most efficient processing: uses least amount of I/O and CPU resources.

Selection of the best method: In a non-procedural language the system does the optimization at the time of execution. On the other hand in a procedural language, programmers have some flexibility in selecting the best method. For optimizing the execution of a query the programmer must know:

- file organization
- record access mechanism and primary or secondary key.
- data location on disk.
- data access limitations.

An Example: significant of query optimization: Consider relations r(AB) and s(CD). We require $r \bowtie s$.

Method 1

a. Load next record e of r in the main memory.
b. Load all records of s, one at a time and concatenate with r.
c. All records of r concatenated? NO: goto a. YES: exit (the result in RAM or on disk).

Performance: Too many accesses.

Method 2: Improvement

a. Load as many blocks of r as possible leaving room for one block of s.
b. Run through the s file completely one block at a time.

Performance: reduces the number of times s blocks are loaded by a factor of equal to the number of r records than can fit in main memory.

Quantification

let n_r: cardinality of r. n_s: cardinality of s.
No. of records of r that can fit in a block = b_r.
No. of records of s than can fit in a block = b_s.
No. of blocks main memory can hold = m.
Total no. of block accesses necessary to read r = n_r/b_r.

Suppose 1 memory block is used for loading blocks from s and m-1 memory blocks are available to r. In performing $r \bowtie s$, s file will be read $\frac{n_r}{b_r\,(m - 1)}$ and each time it will require n_s/b_s block accesses. So we have two parameters (a) the number of times a file block is loaded in RAM blocks and the number of accesses for a file block. Thus, in loading blocks from s file, n_s/b_s blocks will be loaded (accessed) and the total number of times these blocks will be loaded in RAM blocks. The total number of block accesses is then $n_r/b_r + n_r/\, b_r + n_r/\,(b_r\,(m-1))$, that is $\frac{n_r}{b_r}\left(1 + \frac{n_s}{b_s\,(m - 1)}\right)$

If $n_r = n_s = 10{,}000$, $b_r = b_s = 5$ and $m = 100$, the number of accesses required to compute the product = 42,000.

Transfer rate = 20 blocks/sec., then it will take 35 minutes. In reality m will be very large so the time will be much less than 35 mins., but the term is dominated by n_r/b_r and so use the file that has a larger number of blocks in the outer loop.

This example suggests that several points must be considered in the implementation of some of the relational algebra operations. The most expensive operation is $\bowtie$ then $\times$ and then σ. We will consider their implementation with respect to optimization. We discuss the first approach where the system is responsible for selecting the best processing strategy.

Implementation of Join: Join involves two relations. In some cases, a join is a $\times$ or equijoin. We analyze the nature of operations for implementing a join. We will refer to the relations r(AB) and s(CD).

If a query requires that a $\bowtie$ is to be performed then this involves accessing both relations many times as explained earlier. This may be optimized by selecting the smaller relation to be used in the inner loop if both relations cannot fit in memory. We look at a common query:

Query: Find A where B = C and D = 99. RA = $\Pi_A(\sigma_{B = C \text{ and } D = 99}(r \times s))$.

Since D = 99 is meaningful for s, we migrate D= 99 inside (r × s): $\Pi_A(\sigma_{B=C}(r \times \sigma_{D = 99}(s)))$. The second expression is a significant improvement so far as data transfer is concerned. Again, since there is a σ and a $\times$, this can be treated as equijoin with join condition B = C. If we do that then our expression may be written as:

$$\Pi_A(r \bowtie_{B = C} (\sigma_{D = 99}(s))).$$

This expression can be interpreted as follows: It asks for A's that are associated with D = 99. The connecting path between A's and D's is defined by B = C. The above expression cannot be simplified any further. So we try to implement this expression.

Method1: If no index is defined on any of the attributes (ABCD) then the $\bowtie$ can be done first using a sequential search. This will require 2,000 blocks accesses (recall n_s/b_s). From this set we only need C since D is not used in any subsequent condition/operation. This can be done in the memory. The set of C values become set of B value so we can scan r using C values and this will also require 2,000 block accesses (recall n_r/b_r). So the re-written query processing will require 4,000 accesses which is much less than 42,000 accesses.

Method2: If there are indices on all attributes then right blocks are selected and transferred to the memory. This will further reduce the total number of accesses (< 4,000).

Method3: If r and s are large and neither is small enough to fit in memory. In this case we can sort r on B values and s on C values. We can then run through each file once comparing equality. This will require a total of (nr/br + ns/bs) accesses. But the sorting time that extends over m (memory) blocks is proportional to **m log m**, which dominats the time to compare the sorted files.

There may be, depending upon the resource capacity, several other ways of optimizing the processing time of this type of queries.

SELECT: Gives a horizontal subset of a relation. This means testing of each tuple of the relation to check if it satisfies the σ predicate. A predicate may include a simple (SSN = 123456789) or a conjunctive condition (a set of simple conditions connected via AND: Dno=5 AND Name = John) or a disjunctive condition (a set of simple conditions connected via OR: Dno = 5 OR SSN = 123456789). We select the following examples:

Simple conditions: OP1: $\sigma_{SSN=1223456789}$(Employee). OP2: $\sigma_{Dnumber>5}$ (Department)

OP3: $\sigma_{Dno = 5}$ (Employee)

Conjunctive conditions: OP4: $\sigma_{Dno = 5 \text{ AND } Salary > 30000 \text{ AND } Sex = F}$ (Employee)

OP5: $\sigma_{Essn = 1223456789 \text{ AND } Pno = 10}$ (Works_on)

Disjunctive condition: OP4': $\sigma_{Dno = 5 \text{ OR } Salary > 30000 \text{ OR } Sex = F}$ (Employee)

Processing simple conditions

Method1: The simplest method is performing an exhaustive linear search. Each record of a file (sequential, index sequential etc.) is accessed and the condition is verified. This is most expensive, especially if the hit ratio is small.

Examples: OP1, OP2 and OP3

Method2: Use binary search. Condition for this search: records must be sorted on the primary key. This means if no primary key is specified in the condition, a serial search is applied.

Examples: OP1 and OP3.

Method3: Use primary index. Use hashing if home addresses of records are randomized. This method is good for accessing individual record.

Method for non-equality condition

Method4: In this case more than one record is likely to satisfy the condition. Use primary index to get the record that satisfies the equality condition and then access preceding (if comparison involves <) or succeeding (if comparison involves >) records. **Example**: OP2

Method5: Use clustering index. All records having same value of the attribute in the condition (i.e., equality) are grouped together using clustering index.

Method6: Use a secondary B^+ tree index. In the case of equality condition a single record can be accessed. If the tree has been created on a secondary key then a set of records satisfying a condition can be accessed.

Processing Conjunctive conditions

Method7: If there is an access path on the attribute used in the simple condition then retrieve all the records using any method 1 – 6 and select those that satisfy other conditions. Accessing and testing the condition one record at a time will give the same performance.

Method8: Using composite index. If a composite index has been created on the attributes used in the condition then all records satisfying the condition are accessed using the index.

Processing disjunctive conditions: Comparatively harder. This is because a union is required to select the tuples that satisfy the condition. In order to get the correct result, all tuples of both relations must be compared. If anyone of the attributes has no index then a sequential search is the only method.

Heuristic Optimization: In this method relational algebra expressions are expressed in equivalent expressions that take much less time and resource to process. As we illustrated, repositioning relational algebra operations in certain ways does not affect the results. First we present an example to show the effect of this repositioning and then present a list of heuristic rules for optimizing relational algebra expressions. Once an expression is optimized, it can then be implemented efficiently. Consider the following database and query processing

STU	SID	NAME	SMAJOR	**FAC**	FID	AMAJOR	FNAME
	10	James	CS		40	CS	John
	20	Scott	Phy		50	Chem	Place
	--	--	--		--	--	--
	--	--	--		--	--	--

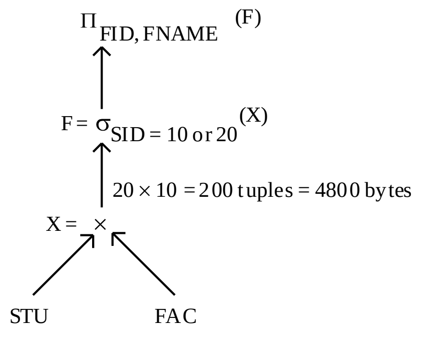
Suppose tuple size in these relations is 24 bytes and STU has 20 tuples and FAC has 10 tuples. STU occupies 24 × 20 = 480 bytes. FAC occupies 10 × 24 = 240 bytes.

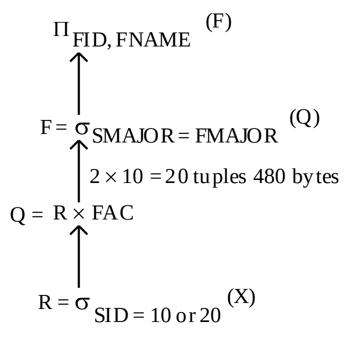
Q: Get name and FID of faculty who advises SID=10 or 20.
RA: $\Pi_{\text{FID, FNAME}} (\sigma_{\text{SID=10 or 20}} (\text{STU} \bowtie \text{FAC}))$

SQL: SELECT FID, FNAME
FROM STU, ADV
WHERE SID = 10 or 20

Query trees for this expression

$\Pi_{\text{FID, FNAME}} (\sigma_{\text{SID=10 or 20}} (\text{STU} \bowtie \text{FAC}))$ $\Pi_{\text{FID, FNAME}} (\sigma_{\text{SID=10 or 20}} (\text{STU}) \bowtie \text{FAC})$





These two queries indicate the saving in performing unary operator first on the relation and then binary operations on other relations.

Repositioning of relational algebra operations: We say two expressions are equivalent if and only if both expressions produce the same results. These transformation rules are given in the text book:

Laws involving $\bowtie$ and $\times$

1. Commutative laws for $\bowtie$ and $\times$: Join condition c and attribute sets A and B, then

$A \bowtie_c B \equiv B \bowtie_c A$. $A \times B \equiv B \times A$.

2. Associative laws for $\bowtie$ and $\times$: If A, B and C are sets of attributes and c1 and c2 are conditions then
   $(A \bowtie_{c1} B) \bowtie_{c2} C \equiv A \bowtie_{c1} (B \bowtie_{c2} C)$ $(A \times B)C \equiv A \times (B \times C)$

Laws involving σ and Π

3. Cascade of σ: $\sigma_{c1}(\sigma_{c2}(r)) \equiv \sigma_{c1}$ and σ_{c2} (r). Since (c1 and c2 = c2 and c1), it follows immediately that σ can be commuted. Therefore, $\sigma_{c1}(\sigma_{c2}(r)) \equiv \sigma_{c2}(\sigma_{c1}(r))$.
4. Cascade of: $\Pi_{A1, A2, ..., An}(\Pi_{B1, B2, ..., Bm}(r)) \equiv \Pi_{A1, A2, ..., An}(r)$. Note that the attributes A1, ..., An, must be among the Bi's for the cascade to be legal.
5. Commuting σ and Π: Selection condition c(X) where X is a set of attributes in c. $X \subseteq A$, where A is the set of attributes for Π. Then $\Pi_{A1, A2, ..., An}(\sigma_c(r)) \equiv \sigma_c(\Pi_{A1, A2, ..., An}(r))$.
6. Commuting σ with × or $\bowtie$: Selection condition c (X) where X is a set of attributes in c and X is also in B (another set of attributes) then $\sigma_c(r \times s) \equiv \sigma_c(r) \times s$.
   a. If c can be expressed as c1 and c2, where $c1 \subseteq A$ and $c2 \subseteq B$ then $\sigma_c(r \times s) \equiv \sigma_{c1}(r) \times \sigma_{c2}(s)$.
   b. If $c1 \subseteq A$ and $c2 \subseteq B$ then also the following equality holds $\sigma_c(A \times B) \equiv \sigma_{c2}(\sigma_{c1}(A) \times B)$. This equivalence allows us to push a part of the selection ahead of ×.
7. Commuting σ and ∪: $\sigma_c(r \cup s) \equiv \sigma_c(r) \cup \sigma_c(s)$.
8. Commuting σ with a -: $\sigma_c(\rho - s) \equiv \sigma_c(r) - \sigma_c(s)$. It should be noted that it is not necessary to have $\sigma_c(s)$ on the right side since $r - s \neq s - r$. So even if we use (s) instead of $\sigma_c(s)$, our expression will be correct. We however, compute this since (s) is larger than $\sigma_c(s)$.
9. Commuting Π and ×: If $A = B \cup C$ then
   $\Pi_{A1, A2, ..., An}(r \times s) = \Pi_{B1, B2, ..., Bm}(r) \times \Pi_{C1, C2, ...Ck}(s)$
10. Commuting Π and ∪: $\Pi_{A1, A2, ..., An}(r \cup s) = \Pi_{A1, A2, ..., An}(r) \cup \Pi_{A1, A2, ..., An}(s)$.

Application of these rules does not guarantee an optimal expression, but it does reduce the processing overheads. As mentioned before, the general idea is to reduce the size of the operands of a binary operator (union, product, intersection etc.) as much as possible. Some special cases occur when the binary operation has operands that are σs and/or Πs applied to leaves of the tree. We must consider carefully how the binary operation is to be done, and in some cases we wish to incorporate the σ and Π with the binary operation. For example, if the binary operation is ∪, we can incorporate σ and Π below it in the tree with no loss of efficiency, as we must copy the operands anyway to form the ∪. However, if the binary operation is ×, with no following σ to make it an equijoin, we would prefer to do σ and Π first, leaving the result in a temporary file, as the size of the operand files greatly influence the time it takes to execute a full ×. We can present this description in the form of an algorithm:

Algorithm: Optimization.	Input:	A relational algebra expression tree.
	Output:	A program for evaluating that expression.

Steps:

1. Use rule 4 to separate each σ, $\sigma_{c1 \text{ and } c2 \ldots cn}(A)$ into the cascade $\sigma_{c1}(\sigma_{c2}(\ldots \sigma_{cn}(A)\ldots))$.
2. For each σ, use rules 4 - 8 to move the selection as far down the tree as possible.
3. For each Π, use rule 3, 9, 10, and the generalized rule 5 to move the Π as far down as possible. Note that rule 4 causes some Πs to disappear, while the generalized rule 5 splits a Π into two Πs, one of which can migrate down the tree if possible. Also, eliminate a Π if it projects an expression onto all its attributes.
4. Use rules 3 - 5 to combine cascades of σs & Πs into a single σ, a single Π, or a σ followed by a Π.
5. Partition the interior nodes of the resulting trees into groups, as follows. Every interior node representing a binary operator $\times$, $\cup$, or - is in a group along with any of its immediate ancestors that are labeled by a unary operator (σ or Π). Also include in the group any chain of descendant labeled by unary operators and terminating at a leaf, except in the case that the binary operator is a $\times$ and not followed by a σ that combines with the $\times$ to form an equijoin.
6. Produce a program consisting of a step to evaluate each group in any order such that no group is evaluated prior to its descendant groups.

Example: Schema

BOOK (TITLE, AUTHOR, PNAME, LIB_NO)
PUBLISHERS (PNAME, PADDR, PCITY)
BORROWERS (NAME, ADDR, CITY, CARD_NO)
LOANS (CARD_NO, LIB_NO, DATE)

We want to execute a query that should give us the following information: **Get the name, address, city, card_no and borrowing data of the borrower and title, author, publisher's name and library of congress no. (LIB_NO), and then list the title of books borrowed before 1/1/82.**

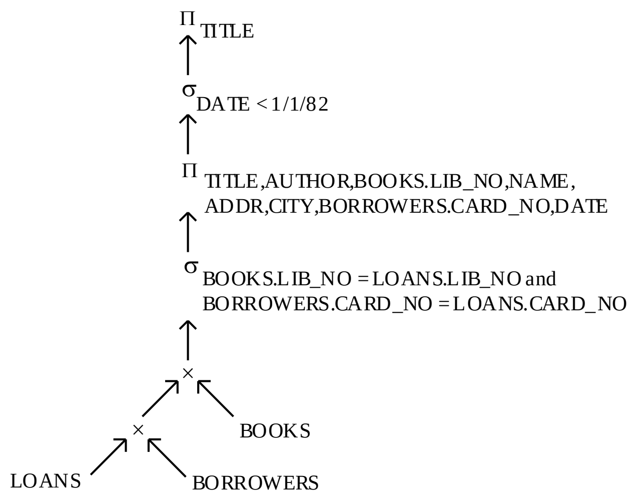
The relational algebra expression:

BLOAN = Π_s (σ_c (LOANS Join BORROWER Join BOOKS))   BTITLE = Π_{TITLE} ($\sigma_{date < 1/1/82}$ (BLOAN))

Where

s = TITLE, AUTHOR, PNAME, LIB_NO, NAME, ADDR, CITY, CARD_NO, DATE
c = BORROWERS.CARD_NO =LOANS.CARD_NO and BOOKS.LIB_NO = LOANS.LIB_NO

Query tree for this expression


Π TITLE
σ DATE < 1/1/82
Π TITLE,AUTHOR,BOOKS.LIB_NO,NAME, ADDR,CITY,BORROWERS.CARD_NO,DATE
σ BOOKS.LIB_NO = LOANS.LIB_NO and BORROWERS.CARD_NO = LOANS.CARD_NO
×
×
BOOKS
LOANS
BORROWERS

Optimization: Split c into two individual conditions a and b:

a. BORROWERS.CARD_NO = LOANS.CARD_NO **b.** BOOKS.LIB_NO = LOANS.LIB_NO

There are three σs so move each of them as far as down the tree as possible. $\sigma_{\text{DATE} < 1/1/82}$ moves below Π and the two σs by rules 4 and 5. This σ then applies to (LOANS × BORROWERS) × BOOKS. Since the DATE is the only attribute mentioned by the σ, and date is an attribute of LOANS, we can replace

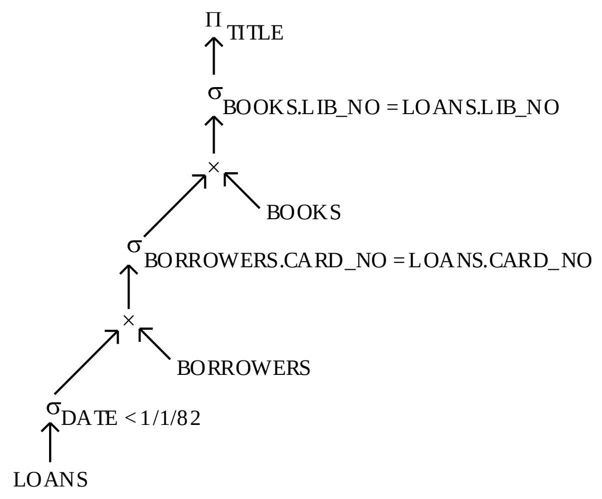
$\sigma_{\text{DATE} < 1/1/82}$ ((LOANS × BORROWER) × BOOKS) by
($\sigma_{\text{DATE} < 1/1/82}$ (LOANS × BORROWER)) × BOOKS and then by
(($\sigma_{\text{DATE} < 1/1/82}$ (LOANS)) × BORROWER) × BOOKS

We have now moved this σ as far down as possible. The σ with condition BOOKS.LIB_NO = LOANS.LIB_NO cannot be moved below either ×, since it involves an attribute of BOOKS and an attribute not belonging to BOOKS. However, the selection on

BORROWERS.CARD_NO = LOANS.CARD_NO can be moved down to apply to
$\sigma_{\text{date} < 1/1/82}$ (LOANS) × BORROWER

Note that LOANS.CARD_NO is the name of an attribute of $\sigma_{\text{date} < 1/1/82}$ (LOANS) since it is an attribute of LOANS, and the result of a Join takes its attributes to be the same as those of the expression to which the Join is applied.

Next, we can combine the two Πs into one, Π_{TITLE}, by rule 3. The resulting tree is shown below.



Π TITLE
σ BOOKS.LIB_NO = LOANS.LIB_NO
×
BOOKS
σ BORROWERS.CARD_NO = LOANS.CARD_NO
×
BORROWERS
σ DATE < 1/1/82
LOANS

Then by extending rule 5 we can replace and by the cascade
$\Pi_{\text{TITLE}}\ \sigma_{\text{BOOKS.LIB\_NO = LOANS.LIB\_NO}}\ \ \Pi_{\text{TITLE, BOOKS.LIB\_NO, LOANS.LIB\_NO}}$

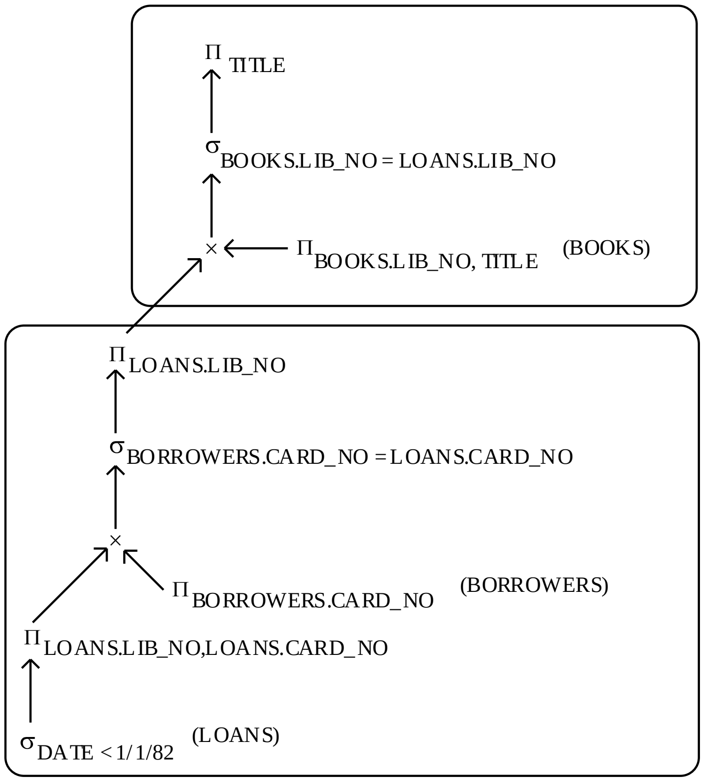
Apply rule 9 to replace the last of these Π_S by $\Pi_{\text{TITLE, BOOKS.LIB\_NO}}$ applied to BOOKS, and $\Pi_{\text{LOANS.LIB\_NO}}$ applied to the left operand of the higher $\times$ in the above figure.

The latter Π interacts with the σ below it by the extended rule 5 to produce the cascade

$\Pi_{\text{LOANS.LIB\_NO}}\ \ \sigma_{\text{BORROWERS.CARD\_NO = LOANS.CARD\_NO}}$

$\Pi_{\text{LOANS.LIB\_NO, BORROWERS.CARD\_NO, LOANS.CARD\_NO}}$

The last of these Πs passes through the $\times$ by rule 9 and passes partially through the σ by the extended rule 5. We then discover that in $\Pi_{\text{LOANS.LIB\_NO, LOANS.CARD\_NO, DATE}}$ the Π is superfluous, since all attributes of LOANS are mentioned. We therefore eliminate this Π. The final tree is shown below.


Π TITLE
σ BOOKS.LIB_NO = LOANS.LIB_NO
×
Π BOOKS.LIB_NO, TITLE (BOOKS)
Π LOANS.LIB_NO
σ BORROWERS.CARD_NO = LOANS.CARD_NO
×
Π BORROWERS.CARD_NO (BORROWERS)
Π LOANS.LIB_NO,LOANS.CARD_NO
σ DATE < 1/1/82 (LOANS)