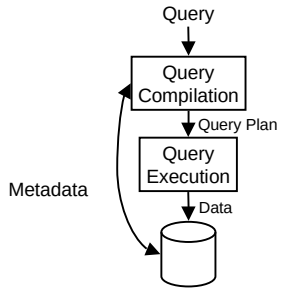
Query Execution

1

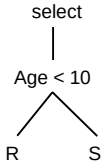
Major Parts of Query Processor

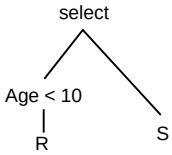


2

From declarative stmts to plans

Select age
from R, S
where R.age < 10 and
R.id = S.id





3

Physical Query Plans

Physical query plans are built from operators, each of which implements one step of the plan.

In what follows, we examine the basic building blocks (operators) of physical query plans.

Usually, an operator is a particular implementation for one of the relational algebra operators

4

Scanning/Sorting Tables

Scanning a relation *R* is the most basic thing we can do…

There are two basic operators we can use:

- *Table-scan*: Used when we know where the blocks of *R* reside
- *Index-scan*: Used when there is an index on any attribute of *R*

We may want to sort *R* while we read it…

Solution: use the *sort-scan* operator…

Possible implementations:

- Use an existing B-tree index on the attribute on which to sort *R*
- If *R* fits in main memory, use *table-scan* or *index-scan* to read *R*, then use a main memory sorting algorithm
- If *R* does not fit in main memory, use multi-way merging!!!

5

Model of Computation

Before we present more complex operations we need a model to evaluate the cost of each operation we use/perform.

- We use the number of disk I/O's as our measure of cost for an operation
- We won't take into consideration the cost, in terms of I/O, of writing the result of a query back to disk.
- Often, we won't even take into consideration the cost of writing intermediate results on disk

6

Parameters for Measuring Cost

There are a number of parameters that are used to express the cost of an operator

- *M* is used to denote the number of main-memory buffers available to an execution of a particular operator
- *B*(R) is used to denote the number of blocks needed to hold all the tuples in relation *R*
- *T*(R) is used to denote the number of tuples in relation *R*
- *V*(*R*, *a*) is used to denote the number of distinct values of the column *a* in *R*

7

Example of Cost Evaluation

Q: What is the cost of a table-scan for relation *R* given that it is *clustered?!?*

A: *B*(R)

Q: What is the cost of a table-scan for relation *R* given that it is not clustered?!?

A: *T*(R)

The above answers are also valid for index-scans…

8

Iterators

Iterators can be used to implement many operators.

An iterator consists of three functions:

- *Open*: It initializes data structures needed to perform the iteration
- *GetNext*: It returns the next tuple in the result and updates the necessary data structures. If there are no more tuples to return, *GetNext* returns a special value *NotFound*
- *Close*: It ends the iteration

Example:

We can use an iterator to implement the table-scan operator

9

Algorithms

Different algorithms can be used to *evaluate* operators…

These can be classified into three categories:

- Sorting-based methods
- Hash-based methods
- Index-based methods

In addition, we can classify them based on their "degree" of difficulty and cost:

- One-pass algorithms
- Two-pass algorithms
- Multi-pass algorithms

10

Operator Classification

- Tuple-at-a-time, unary operators
  e.g. selection, projection
  They can operate on parts of the relation without requiring "*global knowledge"* of the relations
- Full-relation, unary operators
  e.g. grouping, duplicate-elimination
  They require seeing all (or most) of the tuples in the relations
- Full-relation, binary operators
  e.g. union, difference, intersection, joins, products…
  They involve more that one relations

11

One-pass Algorithms for Selection/Projection Operations

Main idea: Given relation *R,* in any of the two operations, we read one block of *R* at a time. Then we perform the operation in each tuple of the block and the selected/projected result tuples are moved to the output buffer (or are pipelined to the next operation).

- Cost in terms of I/O:
  The cost of reading *R* (using table-scan or index-scan)
- Cost in terms of buffer:
  We need at least one buffer page

12

One-pass Algorithms for Duplicate-Elimination Operation

Main idea: Keep in memory one copy of every tuple we have seen

- If we have *M* memory buffers, use one buffer for input, and the rest *M-1* buffers to store tuples we have seen…
- What is the cost of checking whether we have seen a tuple? In general the cost can be n^2
- Cost in terms of I/O and buffer:
  Everything works fine, assuming $B(\delta(R)) =< M$
  Then, the cost in terms of I/O is the cost of reading *R*

13

One-pass Algorithms for Grouping Operation

Main idea: Keep in memory one entry for each group

- If we have *M* memory buffers, we can use one buffer for input, and the rest *M-1* buffers to store group entries…
- Cost in terms of I/O and buffer:
  Difficult to say what is the number of required buffer pages…
  We assume that *M* buffer pages suffice and that the cost in terms of I/O is the cost of reading *R*

14

One-pass Algorithms for Binary Set Operations

Requirement: Given relations *R* and *S*, $min(B(R), B(S)) < M$

Then, we can store the small relation, e.g. *S*, in memory and use one more buffer page to read the other relation, e.g. *R*

- Set Union
  Read *S* into *M-1* buffer pages and also copy them to the output.
  Read *R* one block at a time. Check whether tuple *t* in block of *R* is in *S*… if not, copy *t* to the output…
- Set Intersection
  Read *S* into *M-1* buffer pages. Read *R* one block at a time.
  Check whether tuple *t* in block of *R* is in *S*… If so, output *t*
- Set Difference (*R* – *S*)
  Read *S* into *M-1* buffer pages. Read *R* one block at a time.
  Check whether tuple *t* in block of *R* is in *S*… if so, ignore *t*, else output it…
- Set Difference (*S* – *R*)
  Read *S* into *M-1* buffer pages. Read *R* one block at a time.
  Check whether tuple *t* in block of *R* is in *S*… if so, delete *t* from *S*.
  After considering the whole *R* we copy to the output the tuples of *S* that remain…

15

One-pass Algorithms for Binary Bag Operations

Requirement: Given relations *R* and *S*, $min(B(R), B(S)) < M$

- Bag Union
  Copy each tuple of *R* and each tuple of *S* to the output.
  One buffer page is enough to do that…
- Bag Intersection
  Read *S* into *M-1* buffer pages. Associate with each distinct tuple a *count*…
  Read *R* one block at a time. Check whether tuple *t* in block of *R* is in *S*…
  If so, output *t* and decrement the count of *t* in *S* by one… if *count=0,* drop *t*…
- Bag Difference (*R* – *S*)
  Read *S* into *M-1* buffer pages. Associate with each distinct tuple a *count*…
  Read *R* one block at a time. Check whether tuple *t* in block of *R* is in *S*…
  if not, output *t*, else if *count=0* again output *t*… else, ignore *t* and decrement count…
- Bag Difference (*S* – *R*)
  Read *S* into *M-1* buffer pages. Associate with each distinct tuple a *count*…
  Read *R* one block at a time. Check whether tuple *t* in block of *R* is in *S*… if so, decrement count of *t*… After considering the whole *R* we copy to the output the tuples of *S* that have a positive count…

16

One-pass Algorithms for Product/Natural-Join Operations

Requirement: Given relations *R* and *S*, $min(B(R), B(S)) < M$

- Product
  Read *S* into *M-1* buffer pages. Read *R* one block at a time. For each tuple *t* in block of *R,* concatenate *t* with each tuple of *S.* Output each concatenated tuple…
- Natural Join
  Assume that *R*(*X, Y*) and *S*(*Y, Z*)
  Read *S* into *M-1* buffer pages. Read *R* one block at a time.
  For each tuple *t* in block of *R,* find the tuples in *S* that agree with *t* on all the attributes of *Y*… For each matching tuple, do the join and output the result…

17

Nested-Loop Joins

Before we examine Two-pass algorithms we take a look at some One-and-a-half pass algorithms for the join operator…

- Tuple-based Nested-Loop Join Algorithm (Cost: $T(S) \times T(R)$)

```
FOR each tuple s in S DO
    FOR each tuple r in R DO
        IF r and s join to make a tuple t THEN
            output t
```

- Block-based Nested-Loop Join Algorithm (Cost: $(B(S)+(B(S)/(M-1) \times B(R))$)

```
FOR each chunk of M-1 blocks of S DO
    FOR each block b of R DO
        FOR each tuple t of b DO
            find the tuples of S in memory that join with t
            output the join of t with each of these tuples
```

18

Summary of Algorithms so Far…

Operators	Approximate *M* required	Disk I/O
σ, p	1	*B*
γ, δ	*B*	*B*
∩,∪, -, ×	*min*(*B*(*R*), *B*(*S*))	*B*(*R*) + *B*(*S*)
join	any *M* >= *2*	(*B*(*R*) × *B*(*S*))/*M*

19

Two-pass Algorithms based on Sorting

- Two passes are enough, even for large relations
- Generalizing to more passes can be done easily…

Consider a relation *R*, with *B*(*R*) > *M* (*B*(*R*) <= M^2)
Basic idea behind the algorithms in this category:

- First Pass:
  - Read *M* blocks of *R* into memory
  - Sort these *M* blocks in main memory
  - Write the *sorted sublist* into *M* blocks of disk
- Second Pass:
  Merge the sorted sub-lists that the first pass generated

20

Duplicate-Elimination using Sorting

Assume *M=3,* each block holds two records and relation *R* consists of the following 17 tuples:

2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3

After the first pass the following sorted sub-lists are created:

1, 2, 2, 2, 2, 5
2, 3, 4, 4, 4, 5
1, 1, 2, 3, 5

At the second pass we load each sub-list into a different memory buffer. We consider the first tuple from each block and we find among them the first in sorted order, say *t*. We copy *t* to the output and we remove from all the sub-lists all copies of *t*.

21

Duplicate-Elimination using Sorting

Sub-list	In memory	Waiting on disk
R_1:	1, 2	2, 2, 2, 5
R_2:	2, 3	4, 4, 4, 5
R_3:	1, 1	2, 3, 5

Output 1

Sub-list	In memory	Waiting on disk
R_1:	2	2, 2, 2, 5
R_2:	2, 3	4, 4, 4, 5
R_3:	2, 3	5

Output 2

22

Duplicate-Elimination using Sorting

Sub-list	In memory	Waiting on disk
R_1:	5	
R_2:	3	4, 4, 4, 5
R_3:	3	5

Output 3

Sub-list	In memory	Waiting on disk
R_1:	5	
R_2:	4, 4	4, 5
R_3:	5	

Output 4

23

Duplicate-Elimination using Sorting

Sub-list	In memory	Waiting on disk
R_1:	5	
R_2:	5	
R_3:	5	

Output 5

Sub-list	In memory	Waiting on disk
R_1:		
R_2:		
R_3:		

Finished… What is the cost of the whole algorithm?!?!

$3 \times B(R)$

24

Grouping using Sorting

Main idea similar to duplicate-elimination…

- Create sorted sub-lists of size *M*. Use the grouping attributes as sort key.
- Use one buffer per sub-list
- Find the least value *v* of sort key present among the first available tuples in the buffers. Value *v* becomes the next group for which we
  - Examine all tuples with sort key *v* and compute the needed aggregates
  - When there are no more tuples with value *v* we output a tuple consisting of the grouping attributes plus the value(s) of aggregations
- Cost of the algorithm: $3 \times B(R)$

25

Sort-Based Set Union Algorithm

Consider two relations *R* and *S*…

- Create sorted sub-lists of size *M* both of *R* and for *S*
- Use one buffer per sub-list of *R* and *S*
- Find the first tuple from each block and find among them the first in sorted order, say *t*. We copy *t* to the output and we remove from all the sub-lists all copies of *t*.
- Cost of the algorithm: $3 \times (B(R) + B(S))$
- Additional restriction: $B(R) + B(S) <= M^2$

26

Sort-Based Intersection/Difference

Consider two relations *R* and *S*…

- Create sorted sub-lists of size *M* both of *R* and for *S*
- Use one buffer per sub-list of *R* and *S*
- Find the first tuple from each block and find among them the first in sorted order, say *t*.
  - For set intersection, output *t* if it appears both in *R* and in *S*
  - For set difference (*R* – *S*), output *t* if it appears in *R* but not in *S*
  - For bag intersection, output *t* the minimum number of times it appears in *R* and in *S*
  - For bag difference (*R* – *S*), output *t* the number of times it appears in *R* minus the number of times it appears in *S*
- Cost of the algorithm: $3 \times (B(R) + B(S))$
- Additional restriction: $B(R) + B(S) <= M^2$

27

Sort-Based Join Algorithm

Consider two relations *R*(*X, Y*) and *S*(*Y, Z*)

- Sort both *R* and *S* with *Y* as the sort key (using a two-phase, multi-way merge sort)
- Use only two buffers, one for *R* and one for *S*
- Merge the sorted *R* and *S*
  - Find the value *y* that is at the front of the blocks for *R* and *S*
  - If *y* appears only in one of the relations, drop all tuples with value *y*
  - Otherwise, find ALL tuples from both relations having this value. If necessary, read blocks from *R* and/or *S,* until it is certain that there are no more *y's* in either relation.
  - Output tuples that can be formed by joining tuples from *R* and *S* with a common *y* value
- Cost of the algorithm: $5 \times (B(R) + B(S))$
- Additional restriction: $B(R) <= M^2$ and $B(S) <= M^2$

28

Summary of Sort-Based Algorithms

Operators	Approximate *M* required	Disk I/O
γ, δ	SQRT(B)	3 *B*
∪,∩, -	*SQRT*(*B*(*R*) + *B*(*S*))	3 (*B*(*R*) + *B*(*S*))
join	*SQRT(max(B(R), B(S)))*	5 (*B*(*R*) + *B*(*S*))
join	*SQRT(B(R)* + *B*(*S*))	3 (*B*(*R*) + *B*(*S*))

29

Two-Pass Algorithms based on Hashing

Main idea:

- Hash all tuples using an appropriate hash key Working on one bucket at a time, perform the desired operation

30

Partitioning Relations by Hashing

Use *M-1* buffers to create *M-1* buckets
FOR each block *b* of relation *R* **DO**
read block *b* into the M^{th} buffer
    FOR each tuple *t* in *b* **DO**
        IF buffer for bucket *h*(*t*) is full **THEN**
            copy buffer to disk and initialize new empty buffer for bucket
        copy *t* to buffer for bucket *h*(*t*)
write all non-empty buffers to disk

31

Duplicate-Elimination using Hashing

- Use algorithm from previous slide to create *M-1* buckets
- The hash function used must use all the attributes in the relation in order to do the hashing…
- For each bucket, perform δ on that bucket using an one-pass algorithm...
- As long as $B(R) <= M^2$ the above approach works
- The cost, in terms of I/O, is 3 $B(R)$

32

Grouping/Aggregation using Hashing

- Use presented algorithm to create *M-1* buckets
- The hash function used must use only the grouping attributes in the relation in order to do the hashing…
- For each bucket, perform γ on that bucket using an one-pass algorithm...
- As long as $B(R) <= M^2$ the above approach works
- The cost, in terms of I/O, is 3 $B(R)$

33

Union/Intersection/Difference using Hashing

- Use presented algorithm to create *M-1* buckets for each one of the relations *R* and *S*…
- The hash function used must
  - be the same for both relations and
  - use all the attributes in order to do the hashing…
- For each pair of buckets R_i *and* S_i, perform one-pass bag/set union/intersection/difference
- As long as $min(B(R), B(S)) <= M^2$ the above approach works
- The cost, in terms of I/O, is 3 $(B(R) + B(S))$

34

Hash-Join Algorithm

- Use presented algorithm to create *M-1* buckets for each one of the relations *R* and *S*…
- The hash function used must use the join attributes in order to do the hashing…
- For each pair of buckets R_i *and* S_i, perform one-pass join
- As long as $min(B(R), B(S)) <= M^2$ the above approach works
- The cost, in terms of I/O, is 3 $(B(R) + B(S))$

35

Summary of Hash-Based Algorithms

Operators	Approximate *M* required	Disk I/O
γ, δ	SQRT(*B*)	3 *B*
∪,∩, -	*SQRT*(*B*(*S*))	3 $(B(R) + B(S))$
join	SQRT(*B*(*S*))	3 $(B(R) + B(S))$

36

Index-Based Algorithms

Main idea:
Take advantage of existing indexes, on one (or more) attributes, to perform some of the operations that are of interest

A few things to note:
- Clustered vs. non-clustered relations
- Clustered vs. non-clustered indexes

37

Index-Based Selection

Consider the operation $select_C(R)$
- If there are no indexes on *R* then the cost of the operation is *B*(*R*), in the best case, or *T*(*R*), in the worst case…
- If *C* is *attr* = *value* and there is a (clustered) index on *attr* then the cost of the operation is *B*(*R*) / *V*(*R*, *attr)*
  - Some more I/O might be necessary…
- In the case where the above index is non-clustered, then the cost of the operation is *T*(*R*) / *V*(*R*, *attr)*

If the selection condition *C* retrieves values in a range, then a B-tree can be used to retrieve qualifying tuples efficiently…
If the selection condition *C* is a complex one, then it can be split into a cascade of two selections *C'* and *C''*…

38

Joining by using an Index

Consider two relations *R*(*X, Y*) and *S*(*Y, Z*)
Assume that *S* has an index on the attribute(s) *Y*
- Consider each block *b* of *R* once
- For each block *b*, consider each tuple *t* in *b*
- Let t_y be the value of tuple *t* on the *Y* attribute(s)
- Use index to retrieve the tuples of *S* with which tuple *t* can be joined…
- Cost of the algorithm:
  - To read *R*: *B*(*R*) I/O's, in the best case, or *T*(*R*) I/O's, in the worst case
  - To read *S*: For each tuple of *R* we read an average of *T*(*S*) / *V*(*S*, *Y*) tuples of *S*. If index is clustered the overall cost is *T*(*R*) *B*(*S*) / *V*(*S*, *Y*) If the index is non-clustered then the cost is *T*(*R*) *T*(*S*) / *V*(*S*, *Y*)

If we can use an index to retrieve tuples in sorted order, then we can perform an ordinary sort-join algorithm without having to perform the intermediate step of sorting for (at least) one the relations

39