Query Evaluation

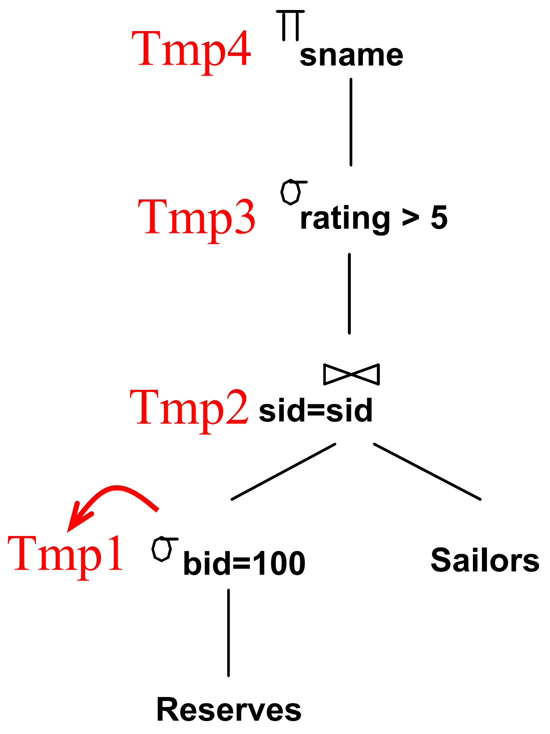
Materialization and Iterators

Query Evaluation Approaches

- Materialization evaluation
  - An operator is evaluated only when each of its operands has been completely evaluated or materialized
  - Intermediate results are materialized to disk
- Pipelining evaluation
  - The output produced by an operator is passed directly to its parent operator
  - Execution of operators is interleaved

Materialization Evaluation

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

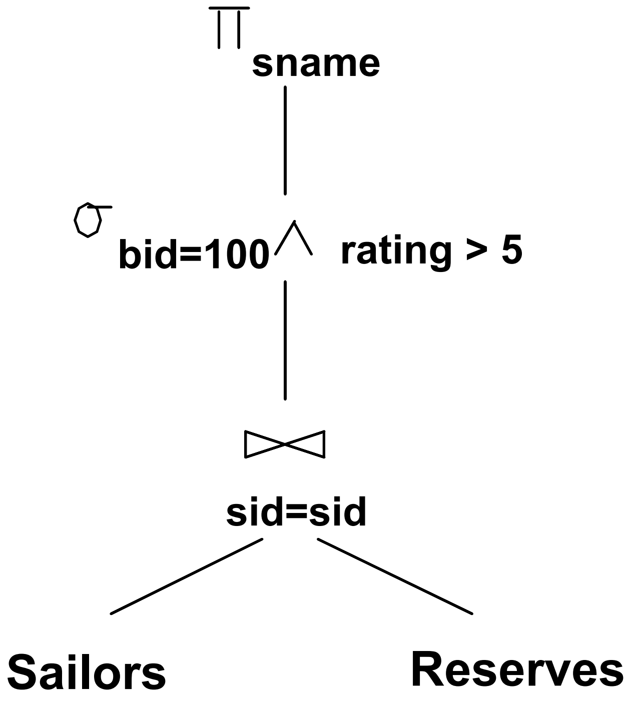


- Tmp1 = Table scan on reserves with bid = 100
- Tmp2 = (Nested-Loops) Join of Tmp1 and Sailors on sid
- Tmp3 = Table scan on Tmp2 with rating > 5
- Result = (Hash-based) Projection of Tmp3 on sname

Iterators for Implementation of Operators

- Most operators can be implemented as an *iterator*
- An iterator allows a consumer of the result of the operator to get the result one tuple at a time

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



Iterators for Implementation of Operators

- Three functions/procedures
  - Open() – starts the process of getting tuples, but does not get a tuple. It initializes any data structures needed
  - GetNext() – returns the next tuple in the result and adjusts the data structures as necessary to allow subsequent tuples to be obtained
    - It may call GetNext() one or more times on its arguments. It also signals whether a tuple was produced or there were no more tuples to be produced
  - Close() – ends the iteration after all tuples have been obtained

```
Open();
While condition is true do {
    GetNext();
    perform other operations
}
Close();
```

An iterator for table-scan operator

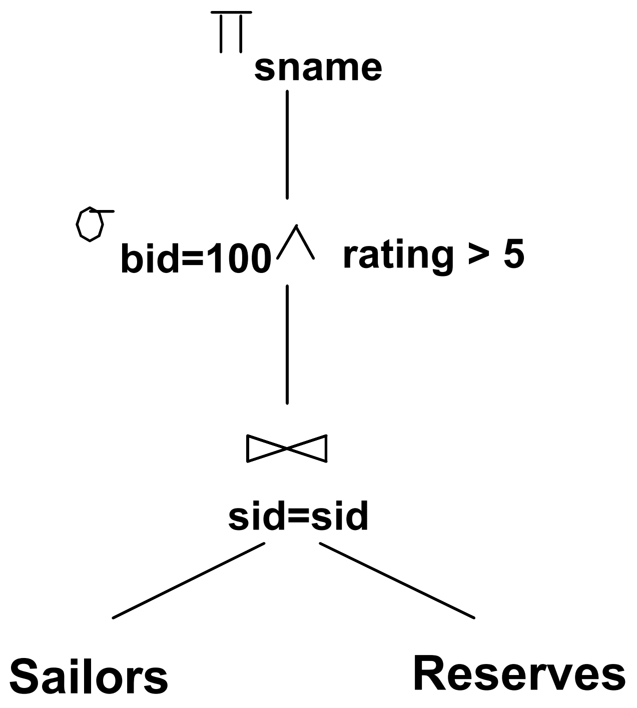
```
Open(R) {
   b := first block of R;
   t := first tuple of block b;
   Found := TRUE;
}

Close(R) {
}
```

```
GetNext(R) {
   If (t is past the last tuple on b)
      b := next block
      If (there is no next block)
            Found := FALSE;
            RETURN;
      Else
            t := first tuple in b;
   oldt := t;
   t := next tuple of b
   RETURN oldt;
}
```

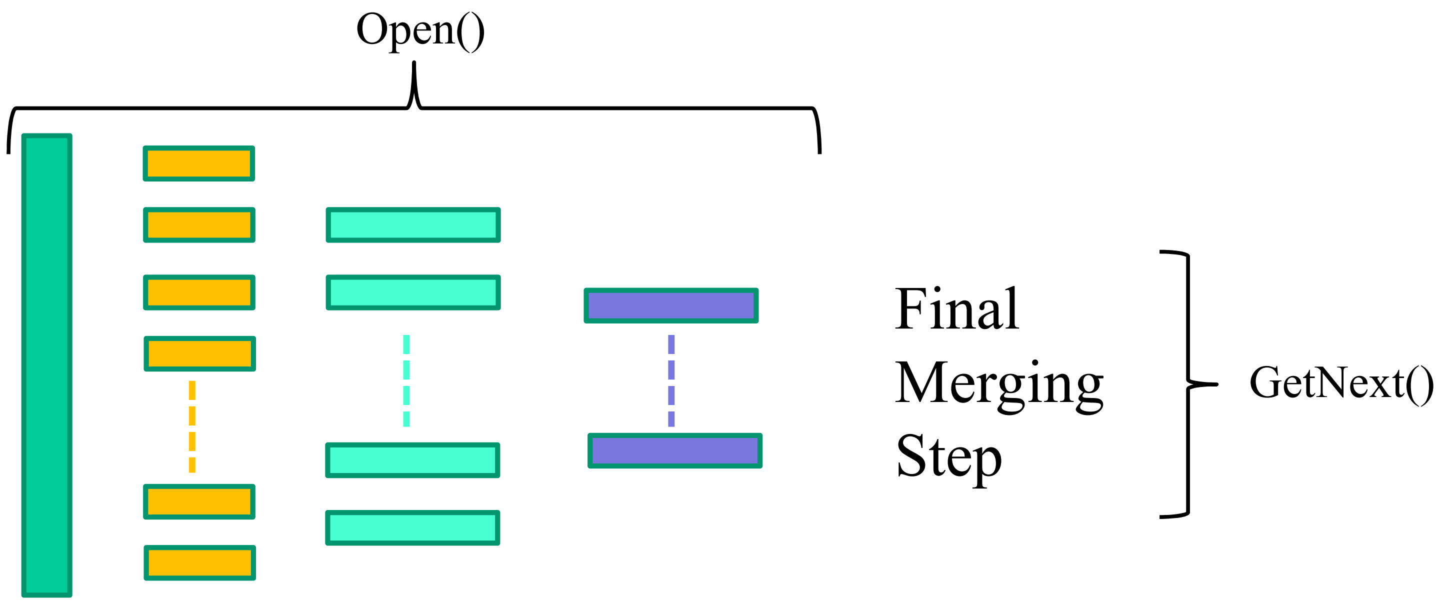
More on Iterators

- Why iterators?
  - Do not need to *materialize* (i.e., store on disk) intermediate results
  - Many operators are *active* concurrently, and tuples flow from one operator to the next, thus reducing the need to store intermediate results


π_{sname}
$\sigma_{\text{bid=100} \wedge \text{rating > 5}}$
$\bowtie_{\text{sid=sid}}$
Sailors
Reserves

More on Iterators

- In some cases (e.g., sort), almost all the work would need to be done by the Open function, which is tantamount to materialization



- We shall regard Open, GetNext, Close as overloaded names of methods
  - Assume that for each physical operator, there is a class whose objects are the relations that can be produced by this operator. If R is a member of such a class, then we use R.Open(), R.GetNext, and R.Close() to apply the functions of the iterator for R

An iterator for tuple-based nested-loops join operator (assumes R and S are non-empty)

```
Open(R,S) {
    R.Open();
    S.Open();
    s := S.GetNext();
}


Close(R,S) {
    R.Close();
    S.Close();
}
```

```
GetNext(R,S) {
    REPEAT
        r := R.GetNext();
        If (NOT Found) {
            R.Close();
            s := S.GetNext();
            IF (NOT Found)
                Return;
            R.Open();
            r := R.GetNext();
        }
    UNTIL (r and s join);
    Return the join of r and s;
}
```

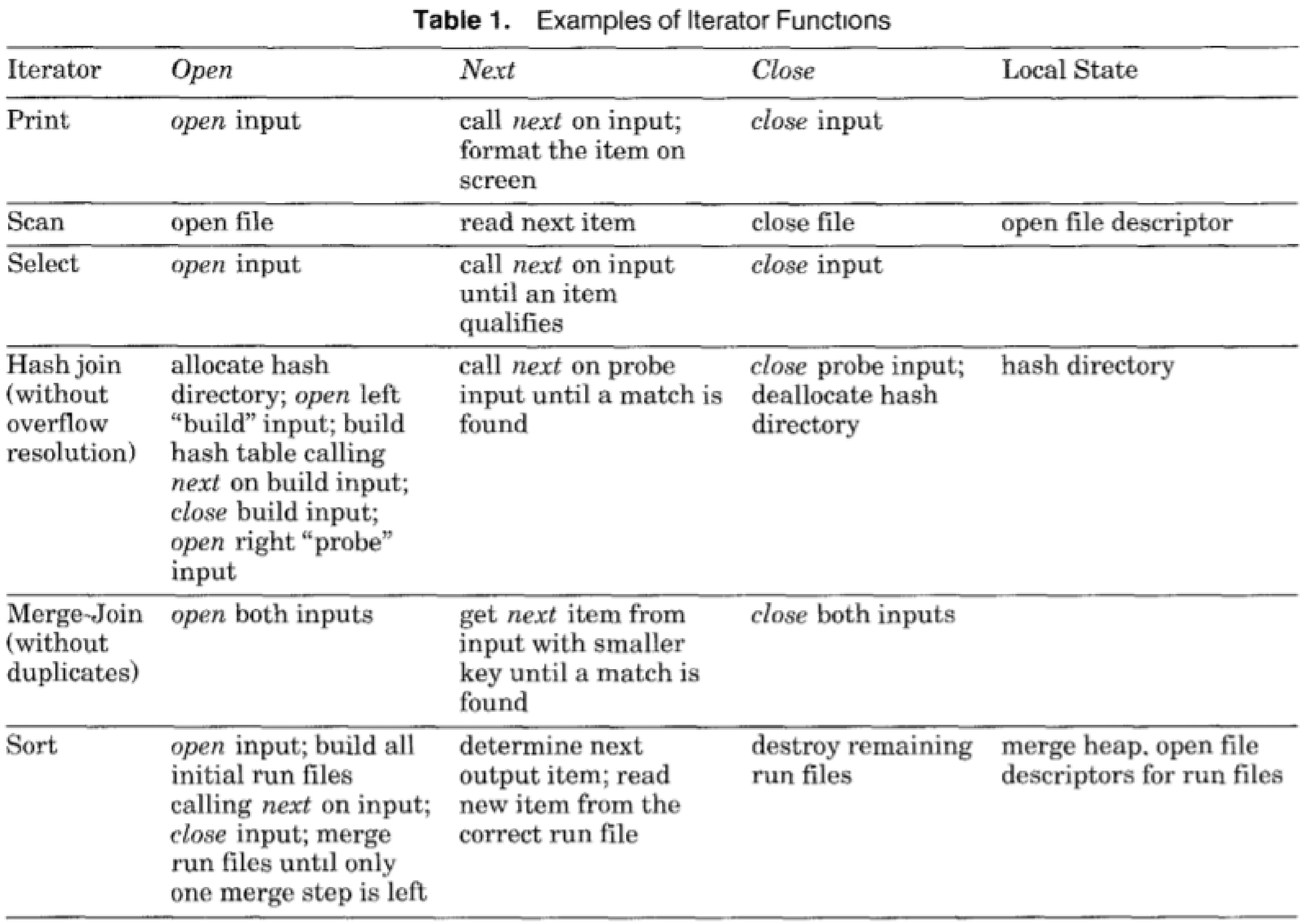


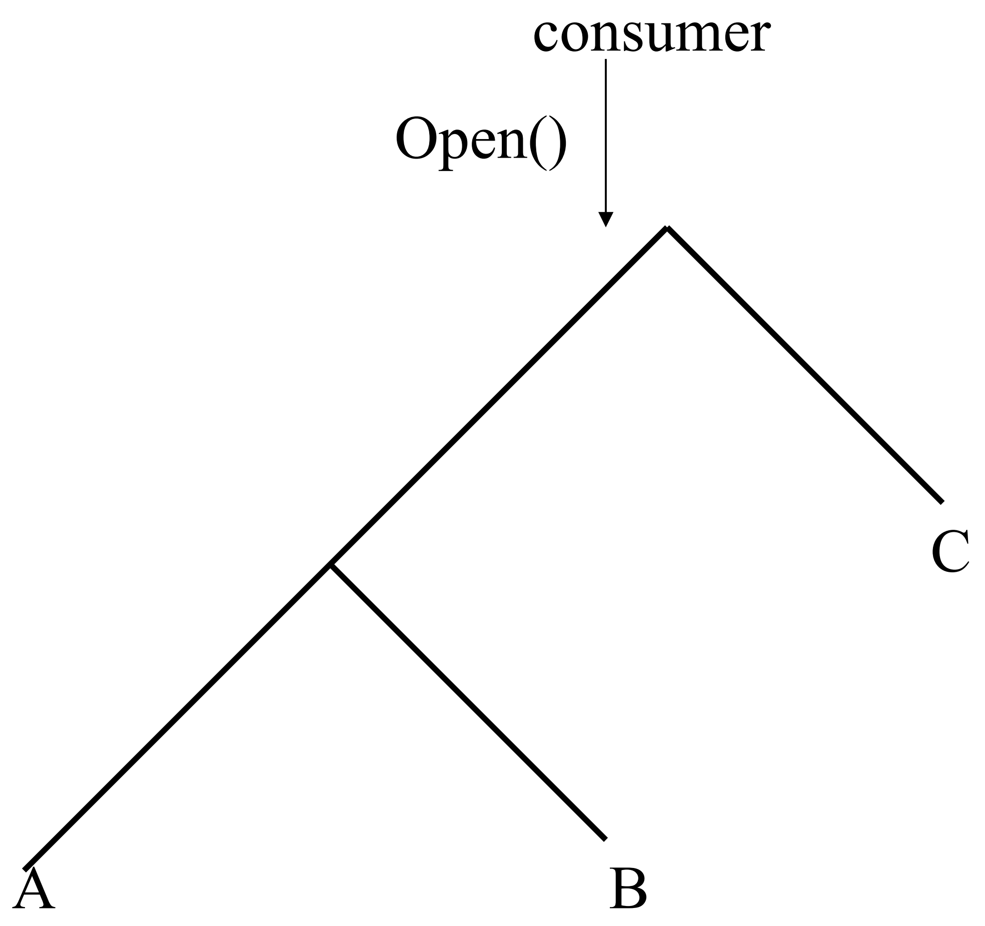
Table 1. Examples of Iterator Functions

Iterator	*Open*	*Next*	*Close*	Local State
Print	*open* input	call *next* on input; format the item on screen	*close* input	
Scan	open file	read next item	close file	open file descriptor
Select	*open* input	call *next* on input until an item qualifies	*close* input	
Hash join (without overflow resolution)	allocate hash directory; *open* left "build" input; build hash table calling *next* on build input; *close* build input; *open* right "probe" input	call *next* on probe input until a match is found	*close* probe input; deallocate hash directory	hash directory
Merge-Join (without duplicates)	*open* both inputs	get *next* item from input with smaller key until a match is found	*close* both inputs	
Sort	*open* input; build all initial run files calling *next* on input; *close* input; merge run files until only one merge step is left	determine next output item; read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

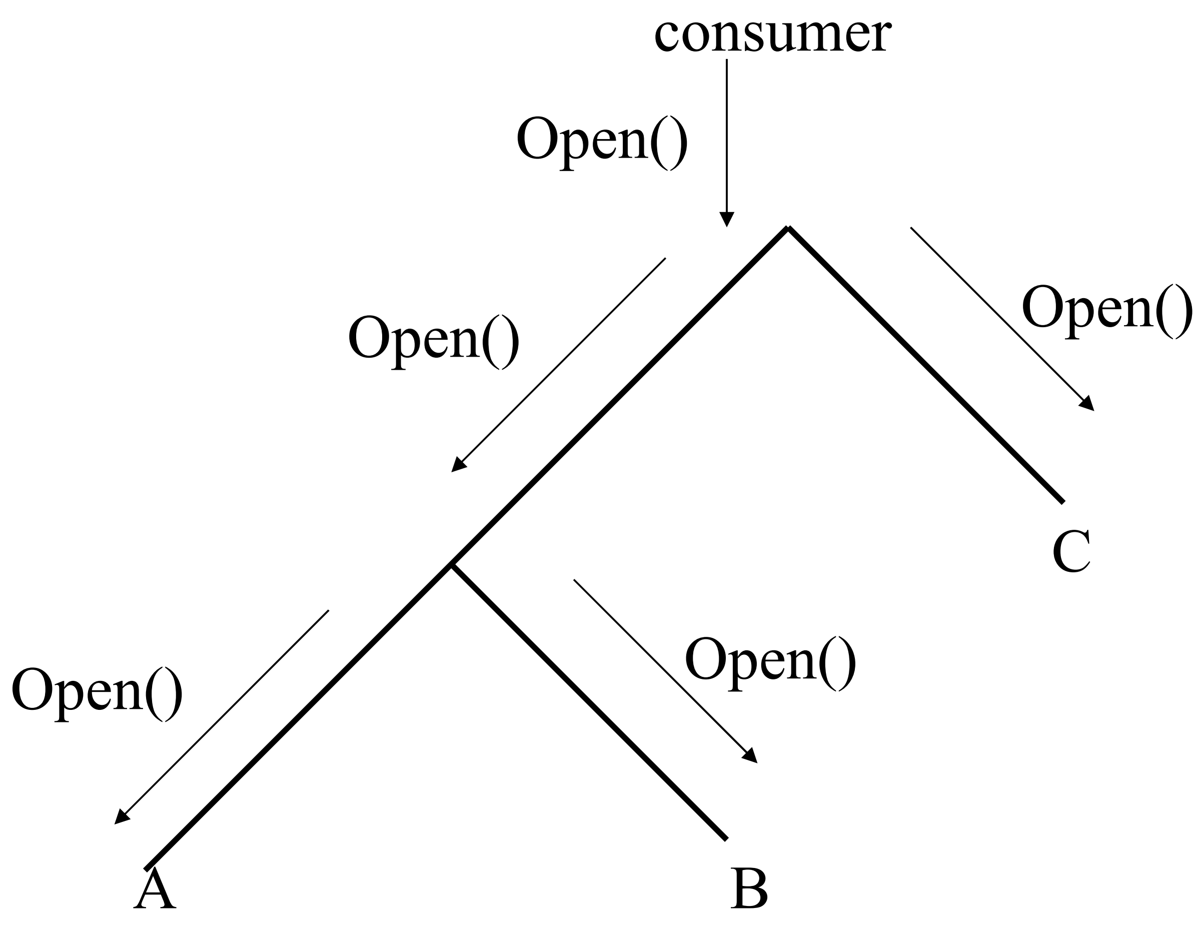
"Query Evaluation Techniques for Large Databases", Goetz Graefe, ACM Computing Surveys, 25:2, Jun 1993

How iterator works in a QEP?

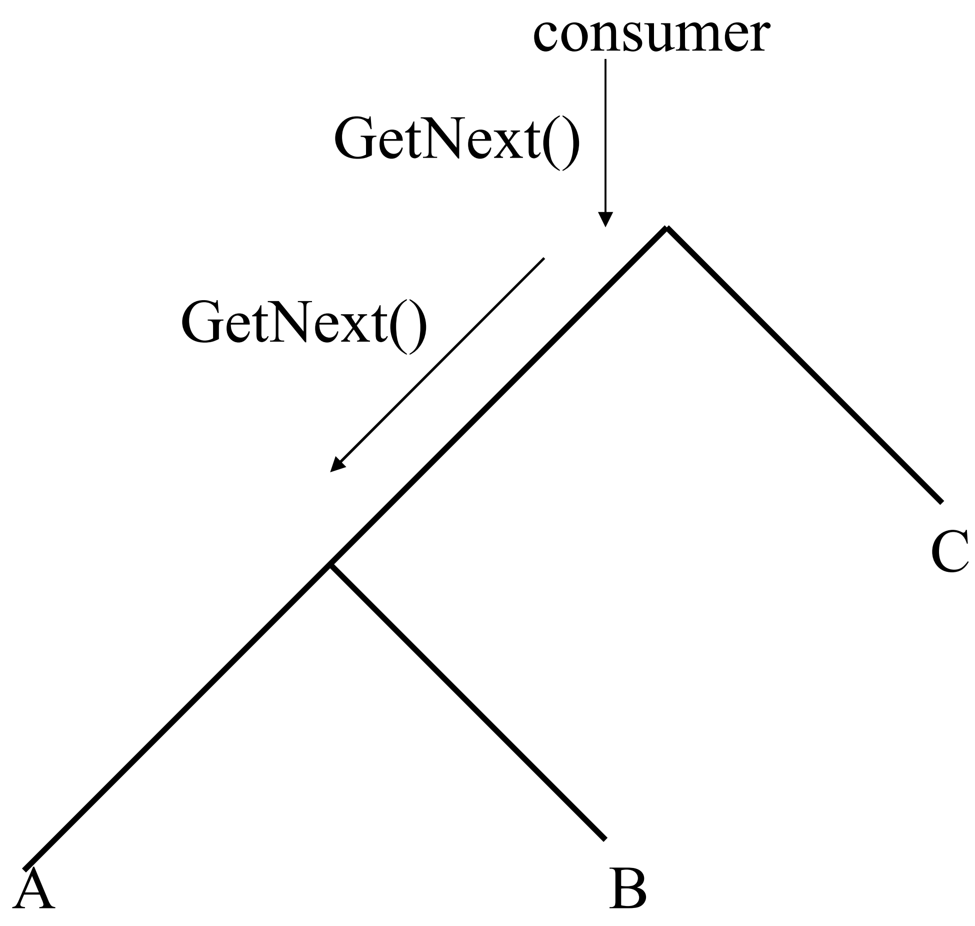
Plan Execution under the Iterator Model



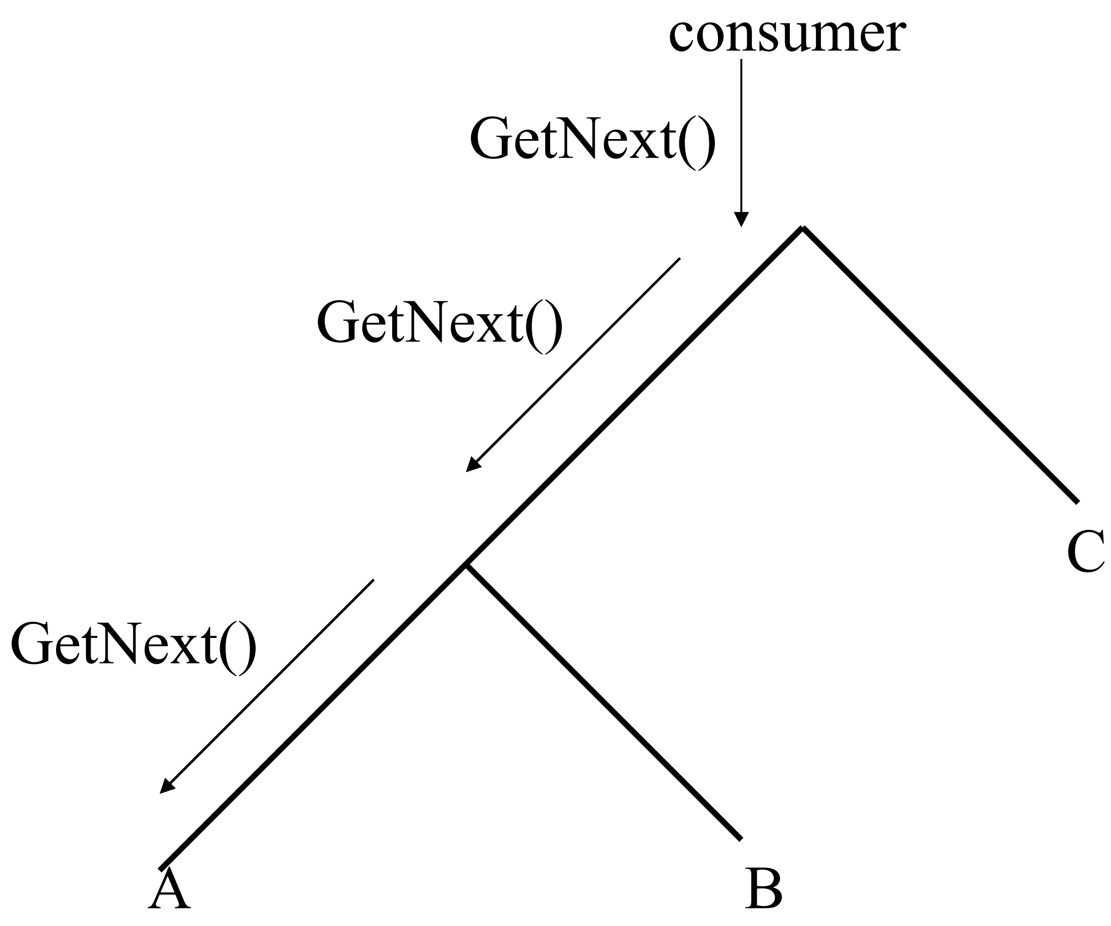
Plan Execution under the Iterator Model



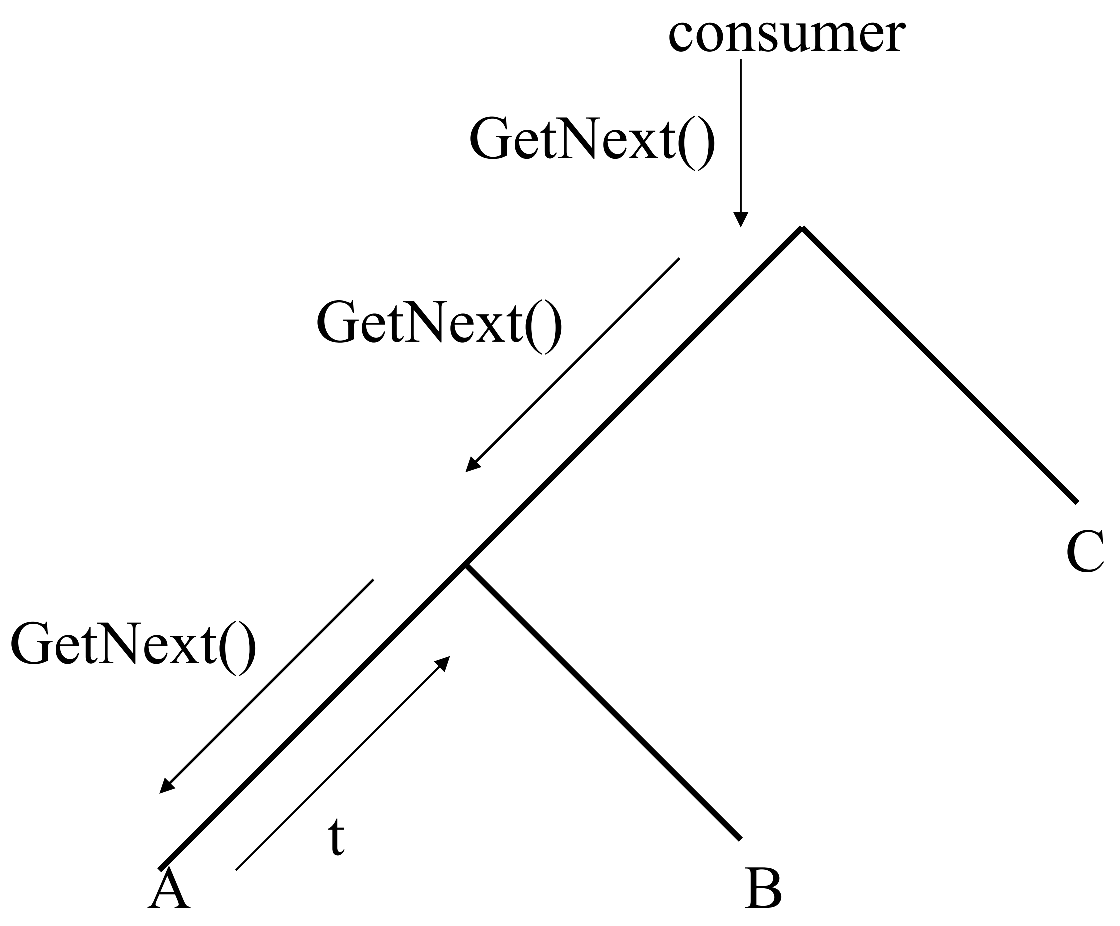
Plan Execution under the Iterator Model



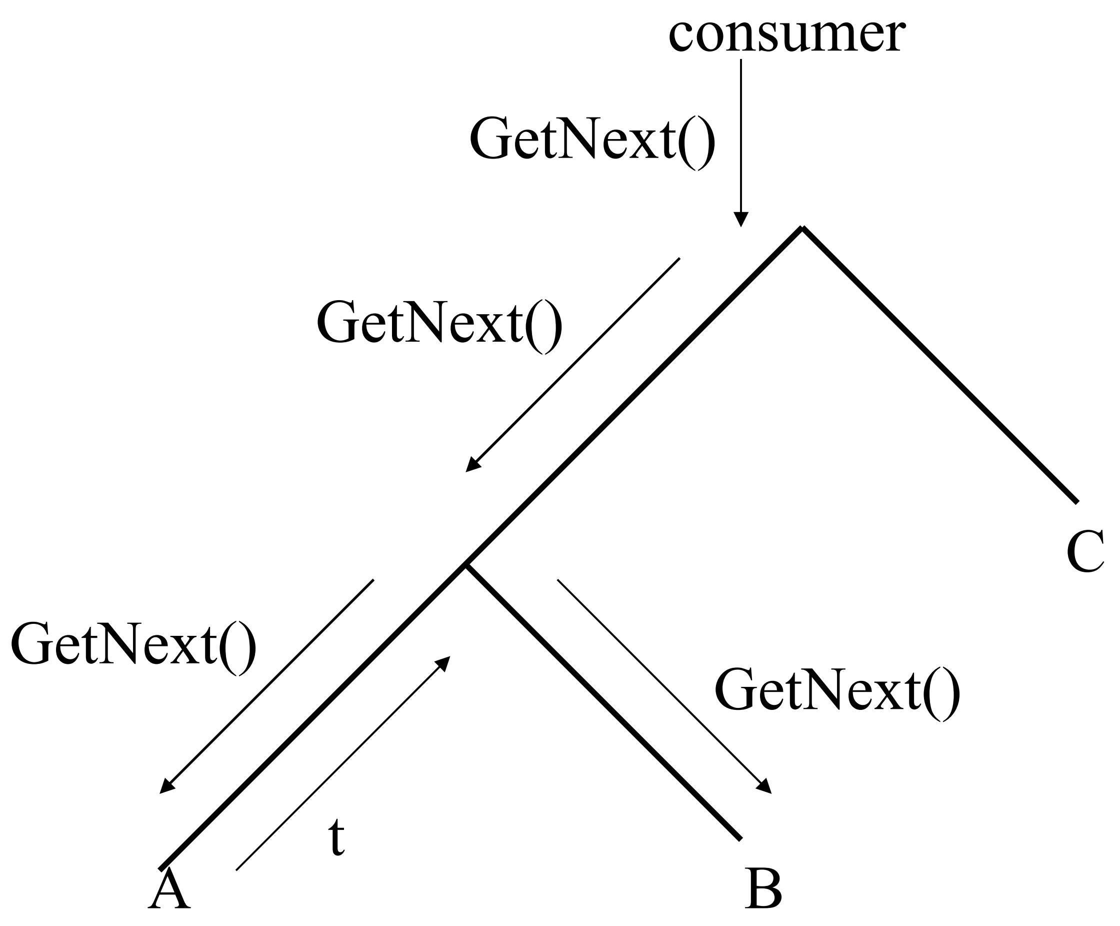
Plan Execution under the Iterator Model



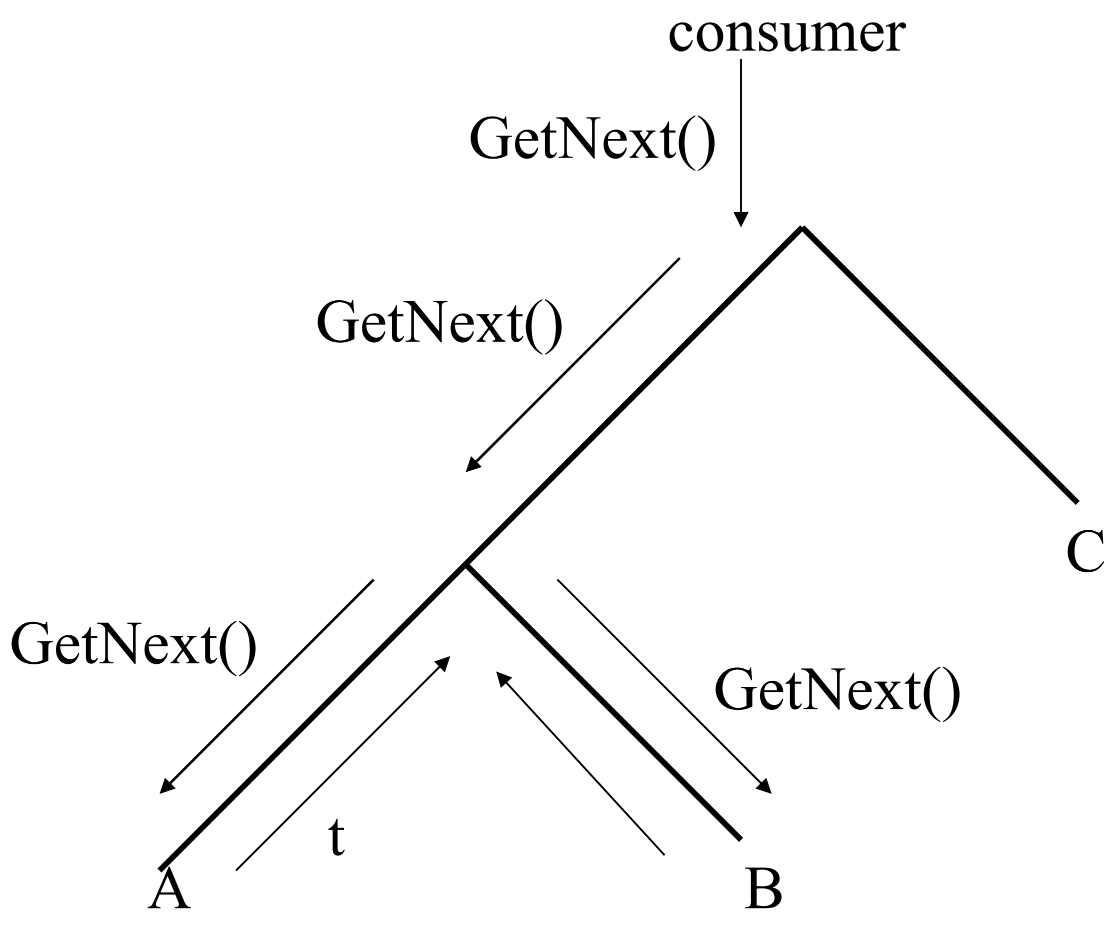
Plan Execution under the Iterator Model



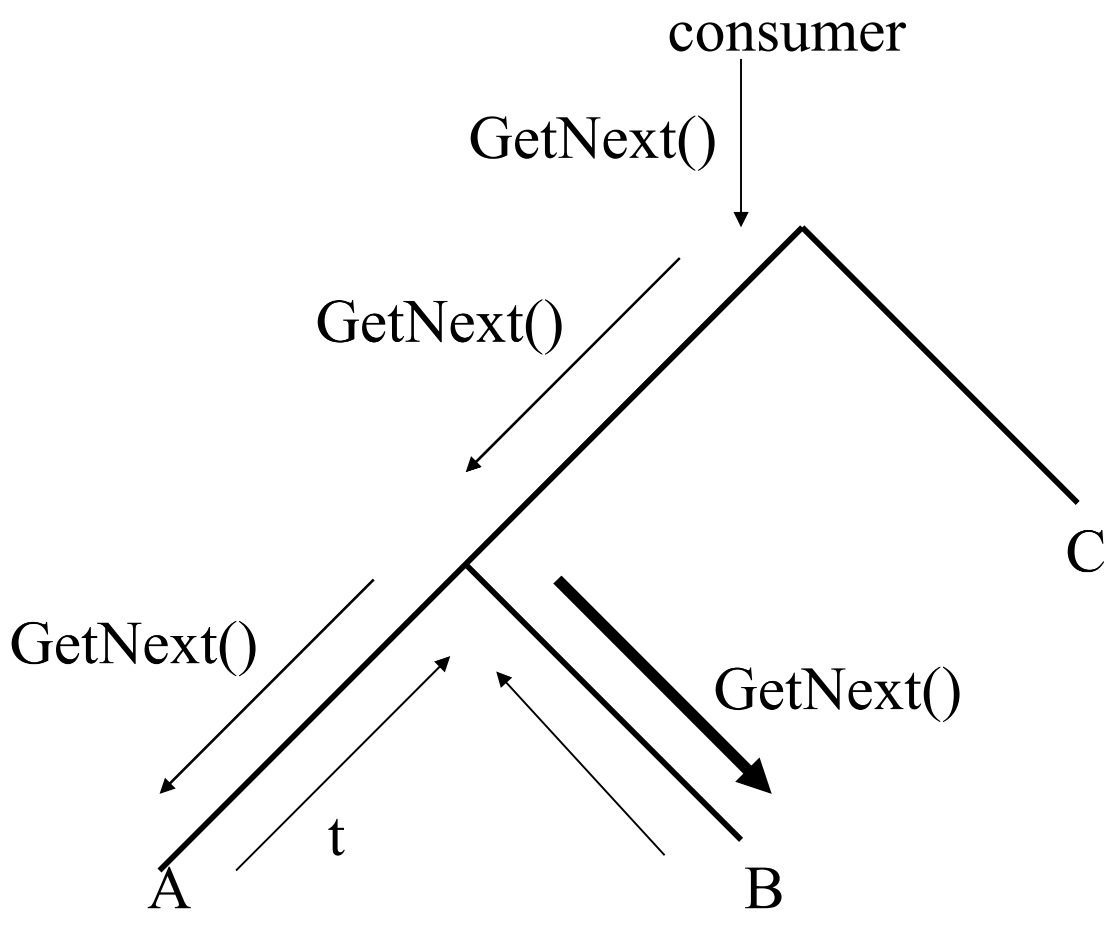
Plan Execution under the Iterator Model



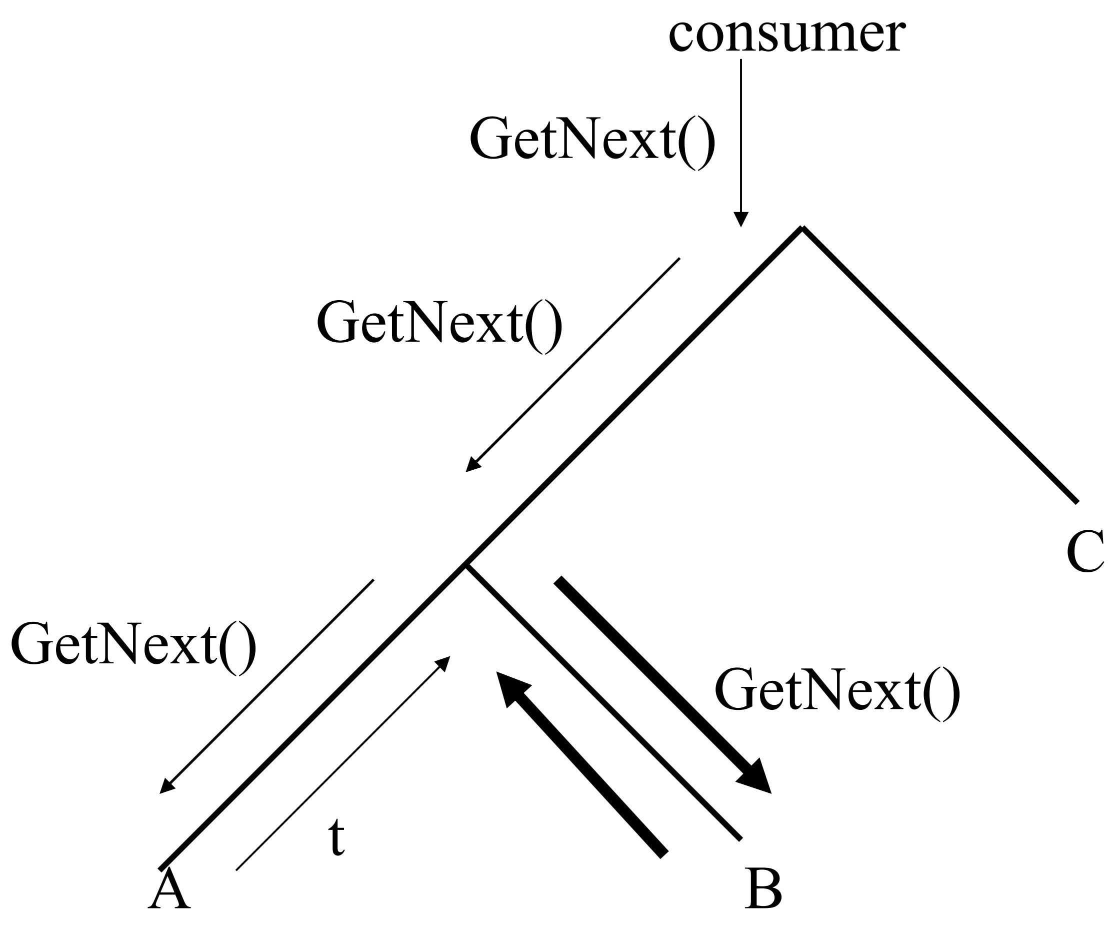
Plan Execution under the Iterator Model



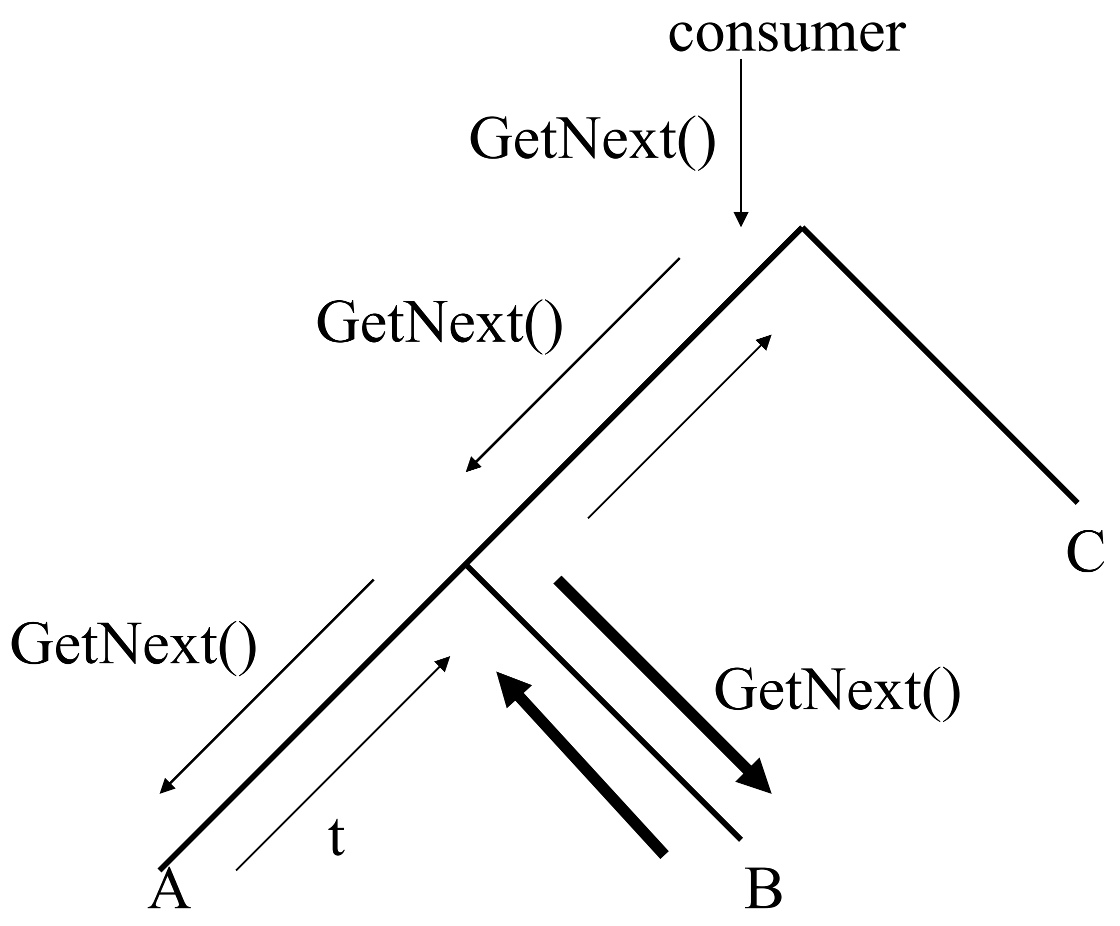
Plan Execution under the Iterator Model



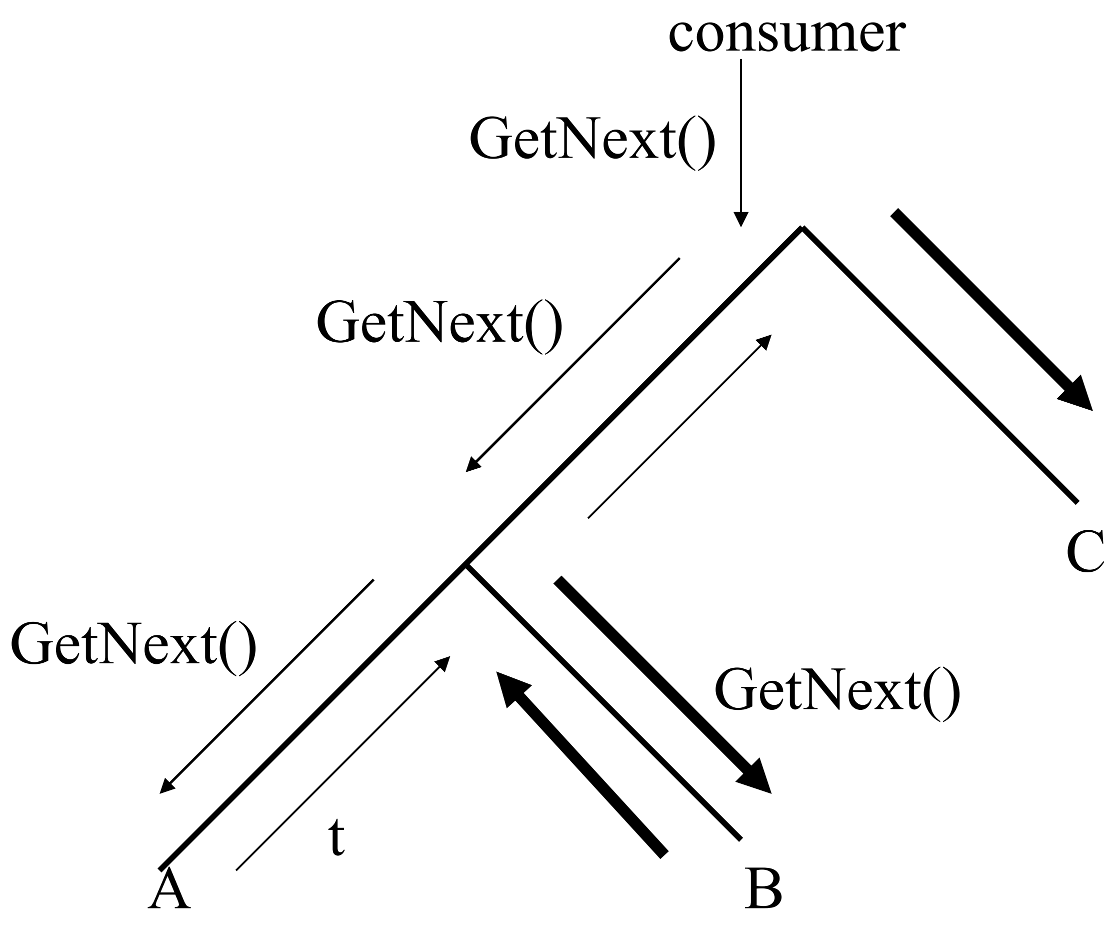
Plan Execution under the Iterator Model



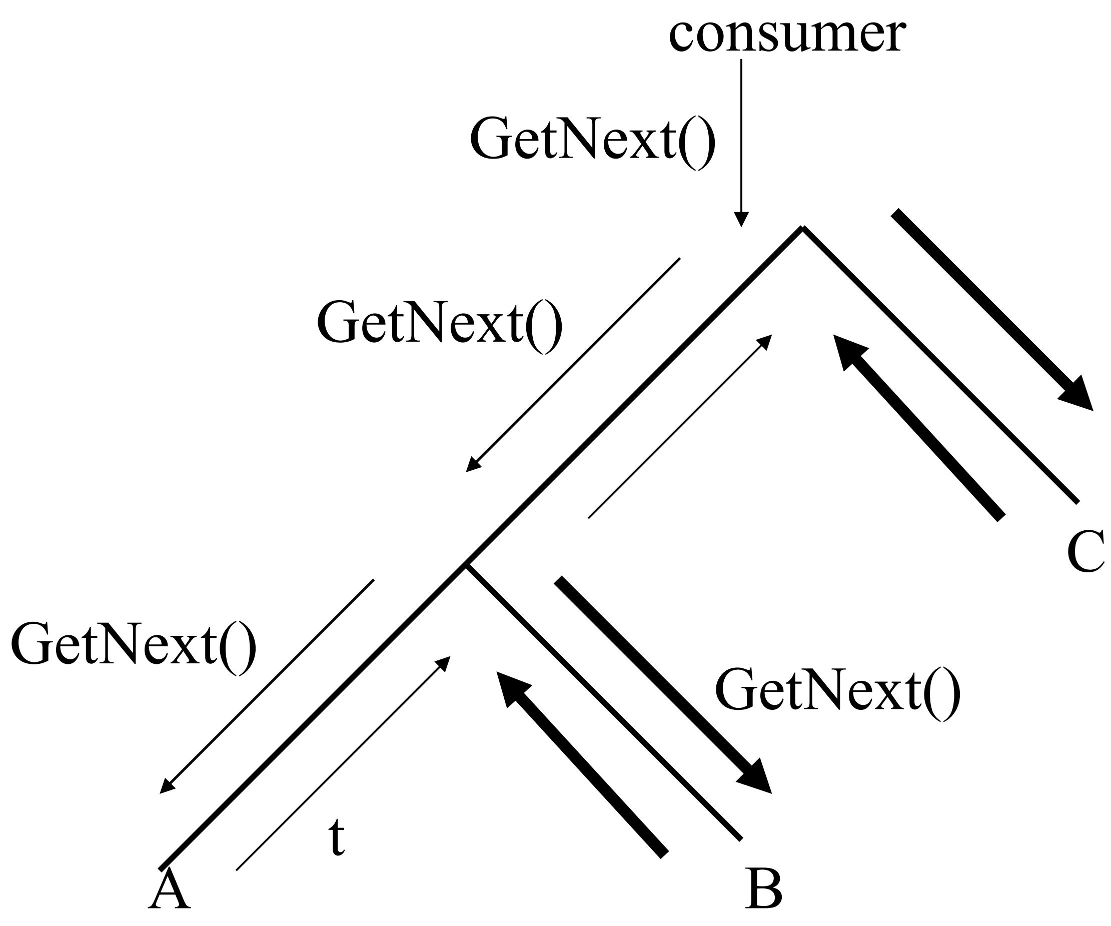
Plan Execution under the Iterator Model



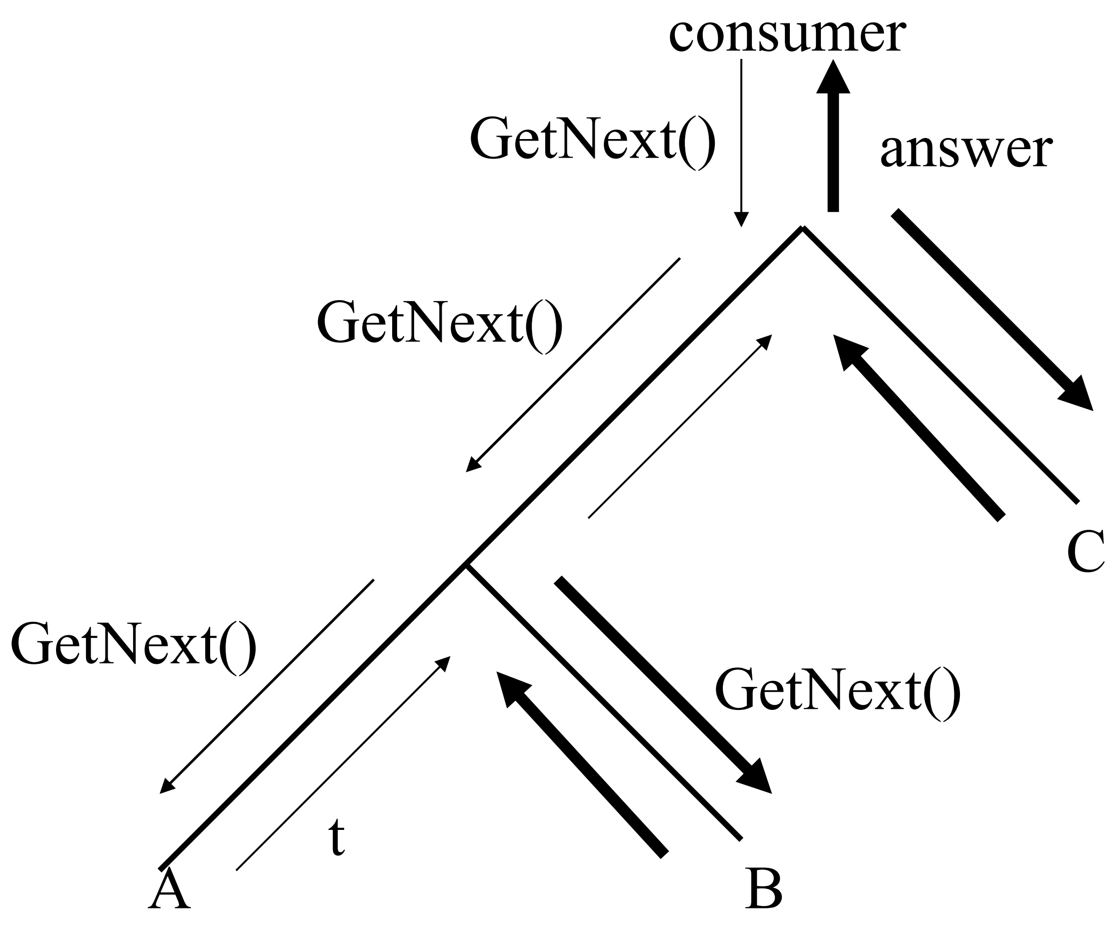
Plan Execution under the Iterator Model



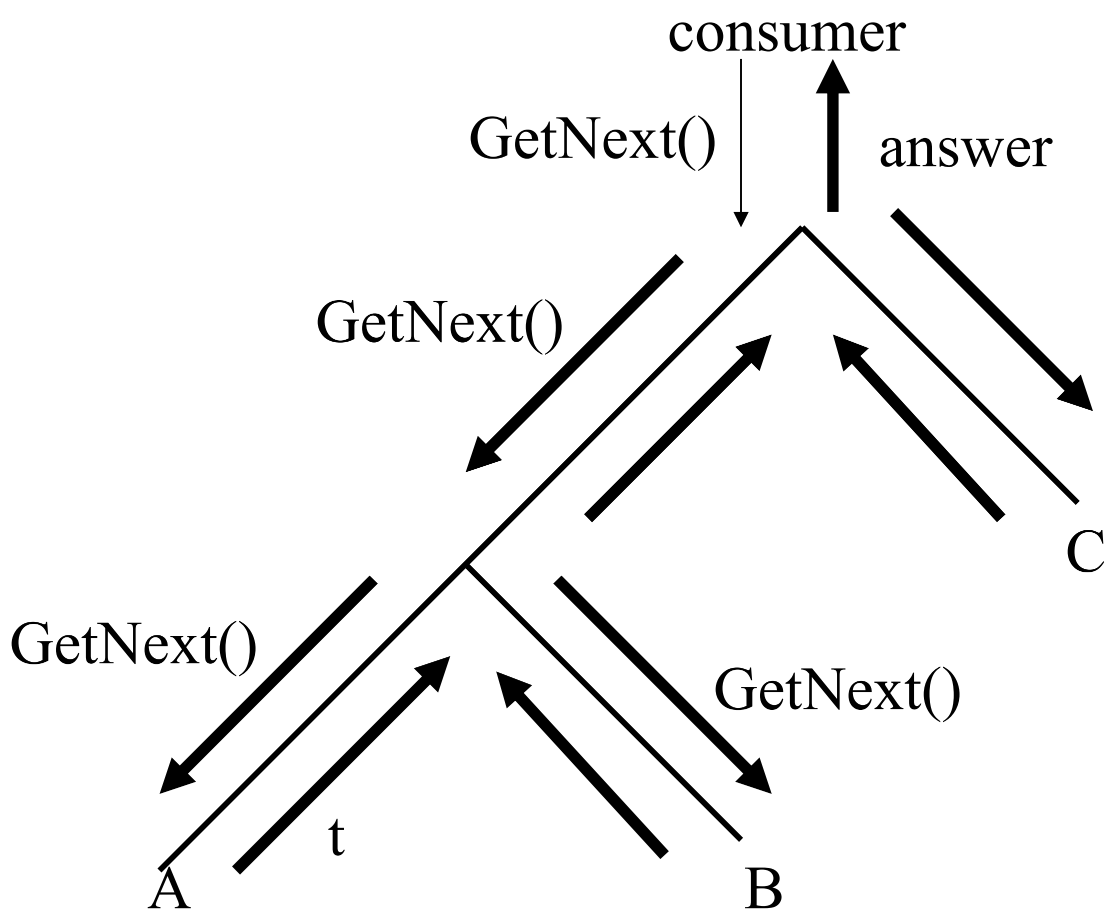
Plan Execution under the Iterator Model



Plan Execution under the Iterator Model



Plan Execution under the Iterator Model



All operators are running simultaneously