Algorithms for query evaluation

Selection, Projection
Queries about data

• Involve two levels of problem-solving:
  • High-level: formulating queries about data (in SQL?)
  • Low-level: implementing algorithms for answering (evaluating) these queries
How do we implement the following query:

```
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```
```sql
SELECT B, D 
FROM R, S 
WHERE R.A = "c" AND S.E = 2 AND R.C = S.C
```
Plan I

- Do **Cartesian product** (produce all pairs FROM R, S)
- **Select** tuples according to WHERE clause
- Do **projection**: select only columns of SELECT clause

```
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```
SELECT B,D
FROM R,S
WHERE R.A = “c” AND S.E = 2 AND R.C=S.C

**Cartesian Product (cross-product)**

<table>
<thead>
<tr>
<th>RxS</th>
<th>R.A</th>
<th>R.B</th>
<th>R.C</th>
<th>S.C</th>
<th>S.D</th>
<th>S.E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each row of R is coupled with each row of S
Scan the resulting table and check conditions

SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C

<table>
<thead>
<tr>
<th>RxS</th>
<th>R.A</th>
<th>R.B</th>
<th>R.C</th>
<th>S.C</th>
<th>S.D</th>
<th>S.E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bingo!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Got one...</td>
<td>c</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
</tbody>
</table>
SELECT B,D
FROM R,S
WHERE R.A = “c” AND S.E = 2 AND R.C=S.C

Plan II

• Do selection on R
• Do selection on S
• Join results on attribute C
• Project B,D columns and place in the result
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C

Plan II

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

\[ \sigma(R) \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

\[ \sigma(S) \]

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>z</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>z</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>x</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>y</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Plan III

Use R.A and S.C Indexes

- Use R.A index to select R tuples with R.A = “c”
- For each R.C value found, use S.C index to find matching tuples from S
- Eliminate S tuples where S.E ≠ 2
- In surviving R,S tuples, project B,D attributes and place in result

```
SELECT B,D
FROM R,S
WHERE R.A = “c” AND S.E = 2 AND R.C=S.C
```
Select B,D
From R,S
Where R.A = “c” AND S.E = 2 AND R.C=S.C

Plan III

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

A=“c”

< c, 2, 10 >
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C

Plan III

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>z</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>y</td>
<td>3</td>
</tr>
</tbody>
</table>
Plan III

```
SELECT B, D
FROM R, S
WHERE R.A = "c" AND S.E = 2 AND R.C = S.C
```
Questions to answer

- What algorithmic techniques are available for each step of query evaluation?
- How do we analyze and compare the cost of each algorithm?
- How do we combine the best-cost algorithms into a larger program?

Several options to evaluate a single operation

\[ \sigma_{\text{name}=\text{Paul}}(\text{student}) \]
- scan file
- use secondary index on \text{student.name}
We consider algorithms for:

- Selection ($\sigma$): select a subset of rows from relation
- Projection ($\pi$): delete unwanted columns from relation
- Join ($\bowtie$): combine two relations according to a given criteria
Select operator

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>x</td>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>y</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

\[ \sigma_{A=B \land D > 5}(R) \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>y</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

`SELECT *`  
`FROM R`  
`WHERE A = B AND D > 5`
Selection algorithm I: 
*one-pass tuple-at-a-time*

- Read the blocks of R one at a time into an input buffer
- Apply select condition to each tuple
- Move selected tuples to the output buffer
Estimating cost

• We use the number of disk I/Os measured in units of 1 block.

• We assume that the input relations for each operator are found on disk, but we exclude the cost of writing an output:
  • Because the cost of writing the output to disk depends on the size of the result, not on the way the result was computed. In other words, it is the same for any computational alternative.
  • Also, we can often pipeline the result to other operators when the result is constructed in main memory. So a final output phase may not even be required.
Cost parameters

- **R**: the name of the relation on disk
- **M**: number of main memory buffers available (1 buffer = 1 block)
- **B(R)**: number of blocks of R
- **T(R)**: number of tuples of R
- **V(R, a)**: number of distinct values in column \( a \) of R
- **V(R, L)**: number of tuples in R that differ by at least one value in the columns listed in L
Cost parameters – contd.

- **R**: the name of the relation on disk
- **B(R)**: number of blocks of R
- **T(R)**: number of tuples of R
- **V(R, a)**: number of distinct values in column *a* of R
- **SC(R, a)**: selection cardinality of *a* in R (average number of matching tuples for each value of *a*)
  - If *a* is a key: \( SC(R, a) = 1 \)
  - If *a* is a non-key: \( SC(R, a) = \frac{T(R)}{V(R, a)} \) (uniform distribution assumption)
- **HT\(_i\)**: number of levels in index *i* (-> height of B-tree)
Selection I: cost

• We scan all B blocks of R
• The cost for Selection I: $B(R)$ disk I/Os
Main algorithmic techniques for improving performance

• Sorting
• Hashing
• Indexes
Selection II: R is sorted on selection condition

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>y</td>
<td>23</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>x</td>
<td>32</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

\[ \sigma_{A=y \land C > 12}(R) \]

```
SELECT *
FROM R
WHERE A = y AND C > 12
```
Selection II: R is sorted

- Do a binary search to locate the first block with tuples satisfying selection condition
- Starting at this block, scan file backward and forward until first encounter of a tuple that does not satisfy the condition
- Add all matching tuples to the output buffer
Selection II: cost

- To find the first block: \( \log_2 B \) disk I/Os
- To retrieve all the qualifying tuples: scan \( SC(R,a) \) tuples:
  - Q: How many blocks for \( SC(R,a) \) tuples?
    - There are \( T/B \) tuples per block
    - Then there are \( SC(R,a) \) \( /[T/B] \) blocks to be scanned
    - \( SC(R,a) = T/V(R,a) \) (assuming uniform distribution)
  - A: Sequential scan of \( B/V(R,a) \) blocks
- Total cost: \( \log_2 B(R) + B(R)/V(R,a) \) disk I/Os
Selection III: R has index on selection condition (or part thereof)

• Search B-tree to find the first qualifying tuple that satisfies the selection condition
• Scan the leaf pages to retrieve all remaining tuples that satisfy the condition
Selection III: cost

• The cost depends on
  • the number of qualifying tuples
  • whether the index is clustered
Clustered Indexes

An index is *clustered* if the underlying data is ordered in the same way as the index’s data entries.
Clustered vs. Unclustered Index

Clustered

Unclustered

Index Entries

Data Records
Clustered vs. Unclustered Index

• Recall that for a disk with block access, **sequential IO is much faster than random IO**

• For exact search, no difference between clustered / unclustered

• For range search over X values: difference between **1 random IO + X sequential IOs**, and **X random IOs**:
  • A random IO costs ~ 10ms (sequential much-much faster)
  • For 100,000 records- **difference between ~10ms and ~17min!**
Selection III: cost

• Finding the first qualifying tuple: $HT_i$
• Assuming that top level is in memory: 1 disk I/O
• If B-tree index is clustered – same as for the sorted file: $B(R)/V(R,a)$
• If B-tree index is unclustered – number of I/Os equals to the number of qualifying tuples – 1 random I/O per tuple: $SC(R, a) = T(R)/ V(R,a)$

Of course in practice we can sort qualifying tuples by RID – to get all tuples in the same block by 1 I/O, but it may well happen that all qualifying tuples belong to different blocks.
Cost estimation exercise

\( \sigma_{a=v}(R) \), and \( B(R) = 1000 \), \( T(R) = 20,000 \)
(20 tuples per block)

- No index on attribute \( a \)
  \( \rightarrow 1000 \) disk I/O’s

- \( R \) has a clustered index on \( a \), \( V(R,a) = 100 \).
  \( \rightarrow 1 + 1000/100 = 11 \) I/O’s

- \( R \) has a non-clustered index on \( a \), \( V(R,a) = 100 \)
  \( \rightarrow 1 + 20,000/100 = 201 \) disk I/O’s

- \( V(R,a) = 20,000 \) (i.e. attribute \( a \) is key)
  \( \rightarrow \) just 2 I/Os

Full scan:
\( B(R) \)
Sorted \( R \):
\( \log_2 B(R) + B(R)/V(R,a) \)
Clustered index on \( R \):
\( HT_i + B(R)/V(R,a) \)
Unclustered index on \( R \):
\( HT_i + T(R)/V(R,a) \)
Selection: complex conditions

Conjunctive: `select * from accounts where balance > 100000 and SIN = "123"`

Disjunctive: `select * from accounts where balance > 100000 or SIN = "123"

- **Option 1**: Sequential scan – always works
- **Option 2 (Conjunctive only)**: Using an appropriate index on one of the conditions
  - E.g. Use SIN index to evaluate SIN = “123”. Apply the second condition to the tuples that match
  - Or do the other way around (if index on balance exists)
  - Which is better?
- **Option 3 (Conjunctive only)**: Use a multi-key index
  - Not commonly available
Selection: complex conditions (contd.)

**Conjunctive:** select * from accounts where balance > 100000 and SIN = “123”

**Disjunctive:** select * from accounts where balance > 100000 or SIN = “123”

• Option 4: Conjunction or disjunction of *record identifiers*
  • Use separate indexes to find all RIDs that match each of the conditions
  • Do an intersection (for conjunction) or a union (for disjunction)
  • Sort the records by block ID and fetch them in one shot
  • Called “Index-ANDing” or “Index-ORing”

- Heavily used in commercial systems
Selection algorithms: summary

- Full scan: scan and match \( B(R) \)
- Sorted \( R \): binary search + sequential scan \( \log_2 B(R) + B(R)/V(R,a) \)
- Clustered index on \( R \): index search + sequential scan \( HT_i + B(R)/V(R,a) \)
- Unclustered index on \( R \): index search + non-sequential retrieval \( HT_i + T(R)/V(R,a) \)

- Space requirements: \( M \geq 1 \) block
### Project operator

**Bag projection – in practice**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>x</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
<td>23</td>
<td>10</td>
</tr>
</tbody>
</table>

**Set projection – in RA theory**

\[ \pi_{A,D}(R) \]

<table>
<thead>
<tr>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>10</td>
</tr>
</tbody>
</table>

**SQL**

```sql
SELECT A, D
FROM R
```

**SELECT DISTINCT**

```sql
SELECT DISTINCT A, D
FROM R
```
Set projection algorithm I: modified 2PMMS

• Sort using $a_1, a_2 \ldots$ as a sorting key

• Phase 1: while reading a single partition, eliminate unwanted attributes – more records per run, tuples are smaller. After sorting in RAM and before writing to disk – remove duplicates (adjacent)

• Phase II: while merging, transfer to output buffer only unique tuples
Set projection algorithm I: diagram

Use to sort a single sublist

Memory M blocks

M buffers

Sorted sublists of R
Set projection algorithm I: cost

• In Phase I, read original relation \( (B) \), write out same number of smaller (less columns, distinct) tuples \( (B) \). Total cost \( 2B \).

• Merge phase: read all \( B \) blocks (at most) of sorted runs (recall: cost of a final writing is not included)

• The total cost of sorting-based projection: \( 3B(R) \) disk I/Os
Set projection algorithm I: memory requirements

- Assuming \( M \) blocks of memory are available, we create sorted runs of size \( \sim M \) each.
- For the second phase, we need 1 block for each run in main memory to a maximum of \( \sim M \) blocks.
- Thus, \( B < M^2 \), and the memory requirement is
  \[
  M \geq \text{sqrt}(B)
  \]
Projection algorithm II: hashing

• Partition tuples into buckets: read R using one input buffer. For each tuple, discard unwanted fields, apply hash function \( h1 \) to choose one of M-1 output buffers.

• When the i-th buffer is full, append its content to one of M-1 on-disk buckets

• Result is M-1 buckets on disk (of tuples with no unwanted fields). 2 tuples from different buckets guaranteed to be distinct (different hash values).

• Read each bucket in turn and build an in-memory hash table, using hash fn \( h2 (<> h1) \) on all fields, while discarding duplicates.

If a set of distinct values from a single bucket does not fit in memory, can apply hash-based projection algorithm recursively to this partition. This may require additional disk I/Os.
Projection algorithm II: diagram

First block holds 1 block of R

\[ h_1(k) = k \mod 3 \]

Memory: M buffers

Memory to process a single bucket
Projection algorithm II: cost

• We read each block of R as we hash the tuples and we write each block to a corresponding bucket for a total of \(2B\) disk I/Os.

• We then read each block of each bucket again in a one-pass algorithm which focuses on the current bucket: \(B\).

• The total number of disk I/Os is \(3B(R)\).
Projection algorithm II: memory requirements

• The cost of 3B(R) can be achieved as long as the individual buckets are sufficiently small to fit in main memory

• Assuming that a good hash function will partition R into equal-sized buckets, each bucket can be approximately \( B/(M-1) \) in size (we have \( M-1 \) output buffers, each writes into its own file)

• If \( B/(M-1) < M \) (fits into memory during individual processing), then the algorithm works with 3B disk I/Os

• Thus, \( M \geq \sqrt{B} \)
Projection III: using indexes

• If an index on the relation contains all wanted attributes in its search key, can do *index-only* scan. Then remove duplicates either by sorting or by hashing.

• If an ordered (i.e., tree) index contains all wanted attributes as *prefix* of a search key, can do even better:
  • Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.
Projection algorithms: summary

- Projection involves duplicate elimination
- This is achieved using 3 main algorithmic techniques:
  - Sorting
  - Hashing
  - Indexing
- Sort-based approach is the standard; better handling of skew and the result is sorted.
Producing output: pipelining vs materialization

- **Materialization**: store the results of each operator on disk until they are needed by another operation

- **Pipelining**: interleave execution of multiple operators
  - The tuples produced by one operator are immediately consumed by another operator, without writing results to disk
  - The operators communicate through the *Iterator* interface
  - For a complex query involving a chain of operators this gives major savings in I/Os

On the other hand, multiple operators share memory, and there is a chance of thrashing
Iterators

• Operators are often implemented as *Iterators*, which allows a consumer of the results to get one resulting tuple at a time

• An iterator has three main methods:
  • **Open**: Initializes data structures. Doesn’t return tuples
  • **GetNext**: Returns next tuple & adjusts the data structures
  • **Close**: Cleans up afterwards

• We assume these to be overloaded names of methods.
Iterator for table-scan of R

_Open_ () {
    b: = the first block of R
    t: = the first tuple of b
}

_GetNext_ () {
    next: = NotFound
    if (t is past the last tuple on block b) {
        increment b to the next block;
        if (there is no next block)
            return NotFound
        else
            t: = the first tuple of b
    }
    next: = t
    increment t to the next tuple of b
    return next
}

_Close_ () {}
Iterator for Selection I (takes as an input GetNext() of table-scan iterator)

\[\text{Open}()\{\]
\]
\[\text{GetNext}()\{\]
\[\quad t: = \text{input.GetNext}()\]
\[\quad \text{next: } = \text{NotFound}\]
\[\quad \text{if } (t \neq \text{NotFound}) \{\]
\[\quad \quad \text{if } (t \text{ satisfies selection condition})\]
\[\quad \quad \quad \text{next: } = t\]
\[\quad \}\]
\[\quad \text{return next}\]
\]
\[\text{Close}()\}
Iterator for Projection II (hashing) Takes as an input table-scan or selection GetNext()

```java
Open () {
    initialize M-1 buckets using M-1 empty output buffers
    t: = input.GetNext()
    while (t != NotFound)
        strip unwanted attributes from t
        if (output buffer h(t) has no room) {
            append content of buffer h(t) to on-disk bucket h(t)
            empty buffer h(t)
        }
        copy t to buffer h(t)
        t: = input.GetNext()
    }
    for each buffer in output buffers
        if (buffer is not empty)
            append buffer to the corresponding on-disk bucket
    ...
```

Part I: partitioning R into M-1 buckets
Iterator for Projection II (contd.)

\(\text{Open}()\) \{ 

... 
initialize \textbf{1 input buffer} to read \(R_0\) 
create empty hash table in the remaining \(M-1\) pages 

\(b: = \) the first block of \(R_0\) 
\(t: = \) the first tuple of \(b\) 

\}

Note: All the preparatory work is done in \textbf{Open}, so we can produce tuple-at-a-time when asked for \textbf{GetNext}
Iterator for Projection II: GetNext

\[ \text{GetNext}() \{ \]
\[ \text{next}: = \text{NotFound} \]
\[ \text{if} \ (t \ \text{is past the last tuple on block } b \ \text{of } R_i) \{ \]
\[ \text{increment } b \ \text{to the next block;} \]
\[ \text{if} \ (\text{there is no next block}) \{ \]
\[ \text{increment } i \ \text{to the next bucket } i+1 \]
\[ \text{if} \ (\text{there is no next bucket}) \]
\[ \text{return } \text{NotFound} \]
\[ \]
\[ \text{empty in-memory hash table} \]
\[ b: = \text{first block of } R_i \]
\[ t: = \text{the first tuple of } b \]
\[ \}
\[ \text{try to insert } t \ \text{into in-memory hash table} \]
\[ \text{while} \ (\text{collision and } t \ \text{is in hash table}) \{ \]
\[ t: = \text{GetNext}() \]
\[ \text{if} \ (t=\text{NotFound} ) \]
\[ \text{return } \text{NotFound} \]
\[ \}
\[ \text{return } \text{NotFound} \]
\[ \]
\[ \text{next}: = t \]
\[ \text{increment } t \ \text{to the next tuple of } b \]
\[ \text{return } \text{next} \]