>>

Implementing Projection

^ >>

❖ The Projection Operation

Consider the query:

```
select distinct name,age from Employee;
```

If the **`Employee`** relation has four tuples such as:

```
(94002, John, Sales, Manager,   32)
(95212, Jane, Admin, Manager,   39)
(96341, John, Admin, Secretary, 32)
(91234, Jane, Admin, Secretary, 21)
```

then the result of the projection is:

```
(Jane, 21)   (Jane, 39)   (John, 32)
```

Note that duplicate tuples (e.g. **`(John,32)`**) are eliminated.

❖ The Projection Operation (cont)

Relies on function **`Tuple projTuple(AttrList, Tuple)`**

- first arg is list of attributes
- second arg is a tuple containing those attributes (and more)
- return value is a new tuple containing only those attributes

Examples, using tuples of type **`(id:int, name:text, degree:int)`**

```
projTuple([id], (1234,'John',3778))
    returns (id=1234)

projTuple([name,degree]), (1234,'John',3778))
    returns (name='John',degree=3778)
```

❖ The Projection Operation (cont)

Without **distinct**, projection is straightforward

```
// attrs = [attr1,attr2,...]
bR = nPages(Rel)
for i in 0 .. bR-1 {
    P = read page i
    for j in 0 .. nTuples(P)-1 {
        T = getTuple(P,j)
        T' = projTuple(attrs, T)
        if (outBuf is full) write and clear
        append T' to outBuf
    }
}
if (nTuples(outBuf) > 0) write
```

Typically, $b_{OutFile} < b_{InFile}$ (same number of tuples, but tuples are smaller)

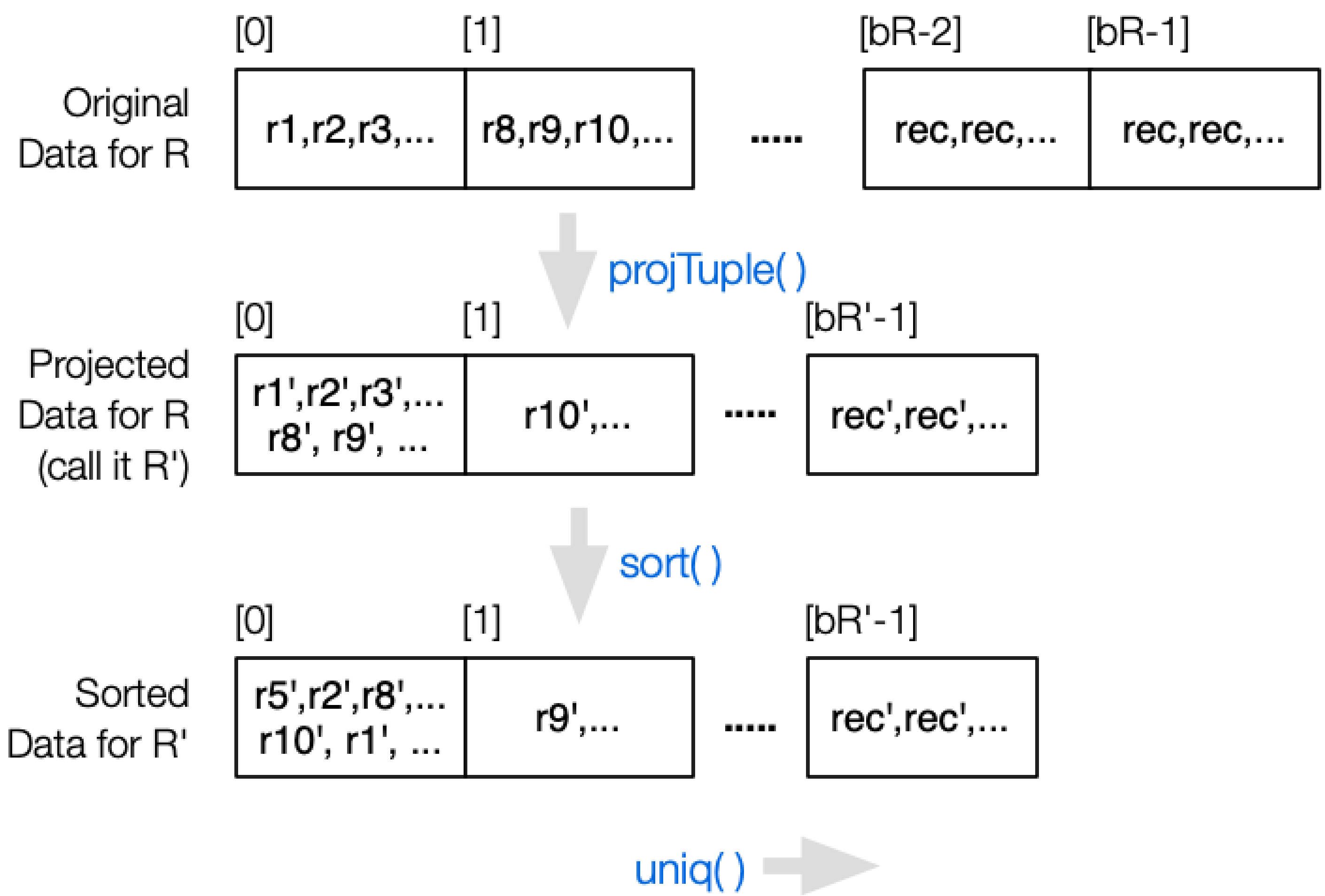
❖ The Projection Operation (cont)

With **`distinct`**, the projection operation needs to:

1. scan the entire relation as input
   - already seen how to do scanning
2. create output tuples containing only requested attributes
   - implementation depends on tuple internal structure
   - essentially, make a new tuple with fewer attributes
     and where the values may be computed from existing attributes
3. eliminate any duplicates produced
   - two approaches: sorting or hashing

<< ^ >>

❖ Sort-based Projection



❖ Sort-based Projection (cont)

Requires a temporary file/relation .

```
for each tuple T in RelFile {
   T' = projTuple([attr1,attr2,...],T)
   add T' to TempFile
}

sort TempFile on [attrs]

for each tuple T in TempFile {
    if (T == Prev) continue
    write T to Result
    Prev = T
}
```

Reminder: "**`for each tuple`**" means page-by-page, tuple-by-tuple

❖ Cost of Sort-based Projection

The costs involved are (assuming $B=n+1$ buffers for sort):

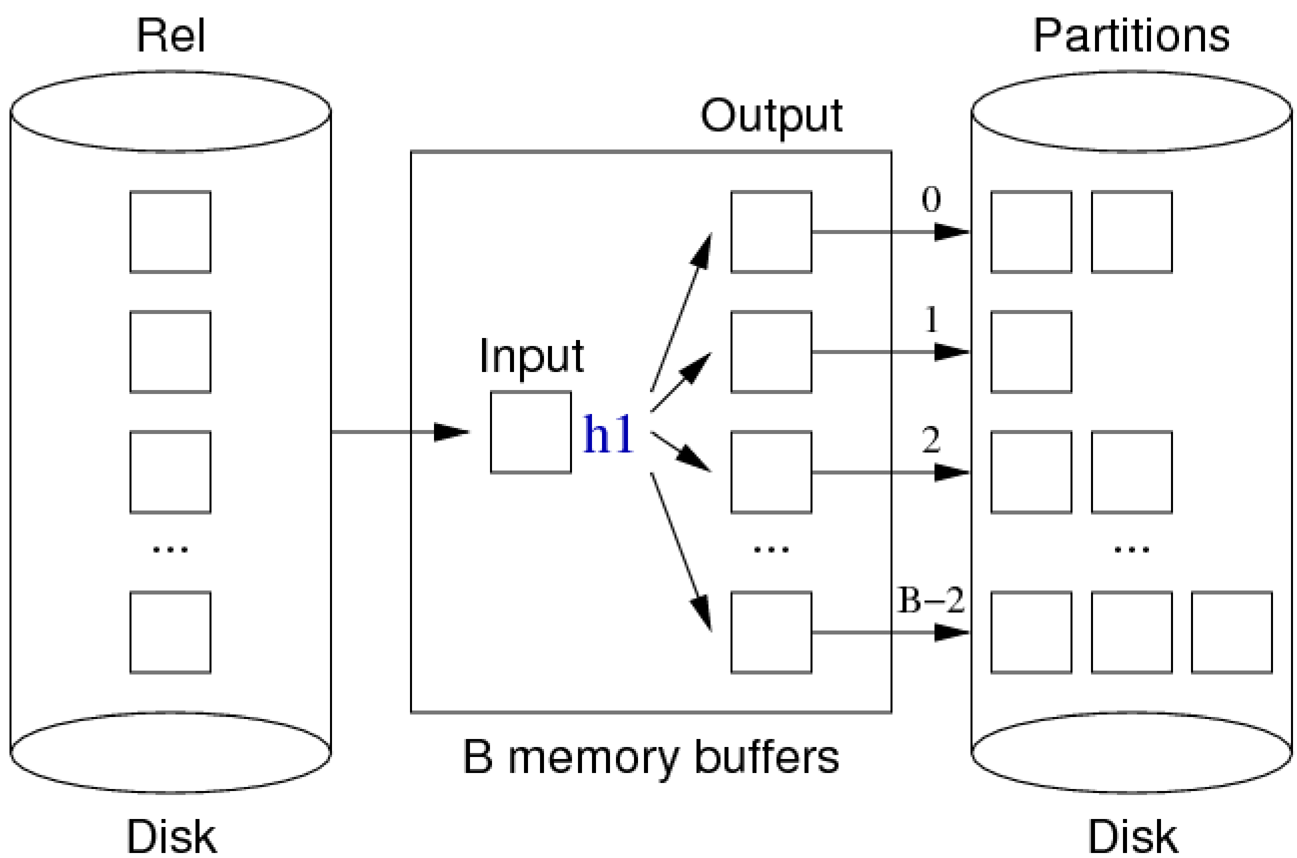
- scanning original relation **`Rel`**: b_R (with c_R)
- writing **`Temp`** relation: b_T (smaller tuples, $c_T > c_R$, sorted)
- sorting **`Temp`** relation:
  $2.b_T.ceil(log_n b_0)$ where $b_0 = ceil(b_T/B)$
- scanning **`Temp`**, removing duplicates: b_T
- writing the result relation: b_{Out} (maybe less tuples)

Cost = sum of above = $b_R + b_T + 2.b_T.ceil(log_n b_0) + b_T + b_{Out}$

<< ^ >>

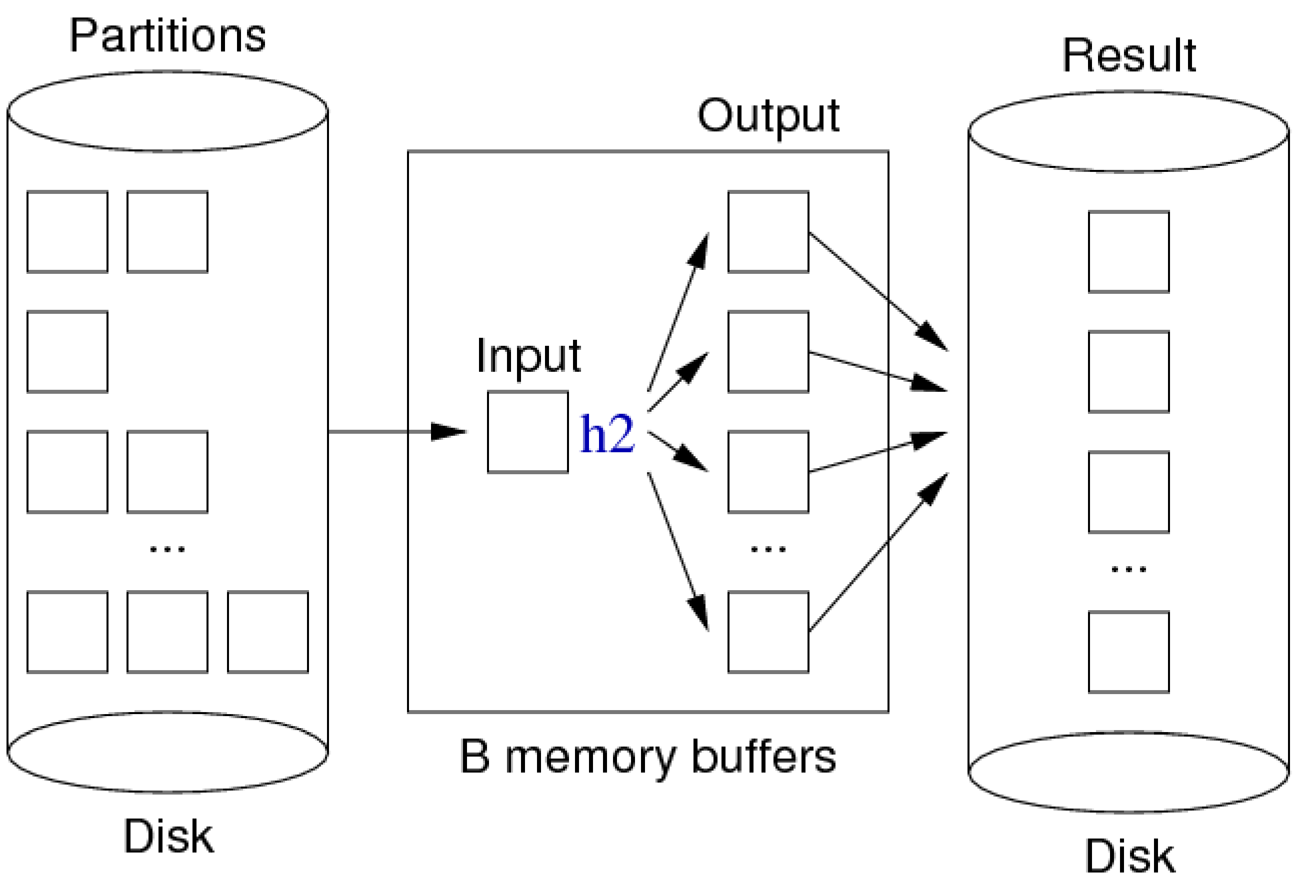
❖ Hash-based Projection

Partitioning phase:



❖ Hash-based Projection (cont)

Duplicate elimination phase:



<< ∧ >>

❖ Hash-based Projection (cont)

Algorithm for both phases:

```
for each tuple T in relation Rel {
    T' = mkTuple(attrs,T)
    H = h1(T', n)
    B = buffer for partition[H]
    if (B full) write and clear B
    insert T' into B
}
for each partition P in 0..n-1 {
    for each tuple T in partition P {
        H = h2(T, n)
        B = buffer for hash value H
        if (T not in B) insert T into B
        // assumes B never gets full
    }
    write and clear all buffers
}
```

Reminder: "**`for each tuple`**" means page-by-page, tuple-by-tuple

❖ Cost of Hash-based Projection

The total cost is the sum of the following:

- scanning original relation **R**: b_R
- writing partitions: b_P (b_R vs b_P?)
- re-reading partitions: b_P
- writing the result relation: b_{Out}

Cost = $b_R + 2b_P + b_{Out}$

To ensure that *n* is larger than the largest partition ...

- use hash functions (h1,h2) with uniform spread
- allocate at least $sqrt(b_R)+1$ buffers
- if insufficient buffers, significant re-reading overhead

❖ Projection on Primary Key

No duplicates, so simple approach from above works:

```
bR = nPages(Rel)
for i in 0 .. bR-1 {
   P = read page i
   for j in 0 .. nTuples(P) {
      T = getTuple(P,j)
      T' = projTuple([pk], T)
      if (outBuf is full) write and clear
      append T' to outBuf
   }
}
if (nTuples(outBuf) > 0) write
```

<< ^ >>

❖ Index-only Projection

Can do projection without accessing data file iff ...

- relation is indexed on $(A_1,A_2,...A_n)$ (indexes described later)
- projected attributes are a prefix of $(A_1,A_2,...A_n)$

Basic idea:

- scan through index file (which is already sorted on attributes)
- duplicates are already adjacent in index, so easy to skip

Cost analysis ...

- index has b_i pages (where $b_i \ll b_R$)
- Cost = b_i reads + b_{Out} writes

❖ Comparison of Projection Methods

Difficult to compare, since they make different assumptions:

- index-only: needs an appropriate index
- hash-based: needs buffers and good hash functions
- sort-based: needs only buffers ⇒ use as default

Best case scenario for each (assuming *n+1* in-memory buffers):

- index-only: $b_i + b_{Out} \ll b_R + b_{Out}$
- hash-based: $b_R + 2.b_P + b_{Out}$
- sort-based: $b_R + b_T + 2.b_T.ceil(log_n b_0) + b_T + b_{Out}$

We normally omit b_{Out} ... each method produces the same result

<< ∧

❖ Projection in PostgreSQL

Code for projection forms part of execution iterators:

- include/nodes/execnodes.h
- backend/executor/execQual.c

Types:

- `ProjectionInfo { type, pi_state, pi_exprContext }`
- `ExprState { tag, flags, resnull, resvalue, ... }`

Functions:

- `ExecProject(projInfo,...)` ... extracts projected data
- `check_sql_fn_retval(...)` ... evaluates attribute value

Produced: 6 Mar 2021