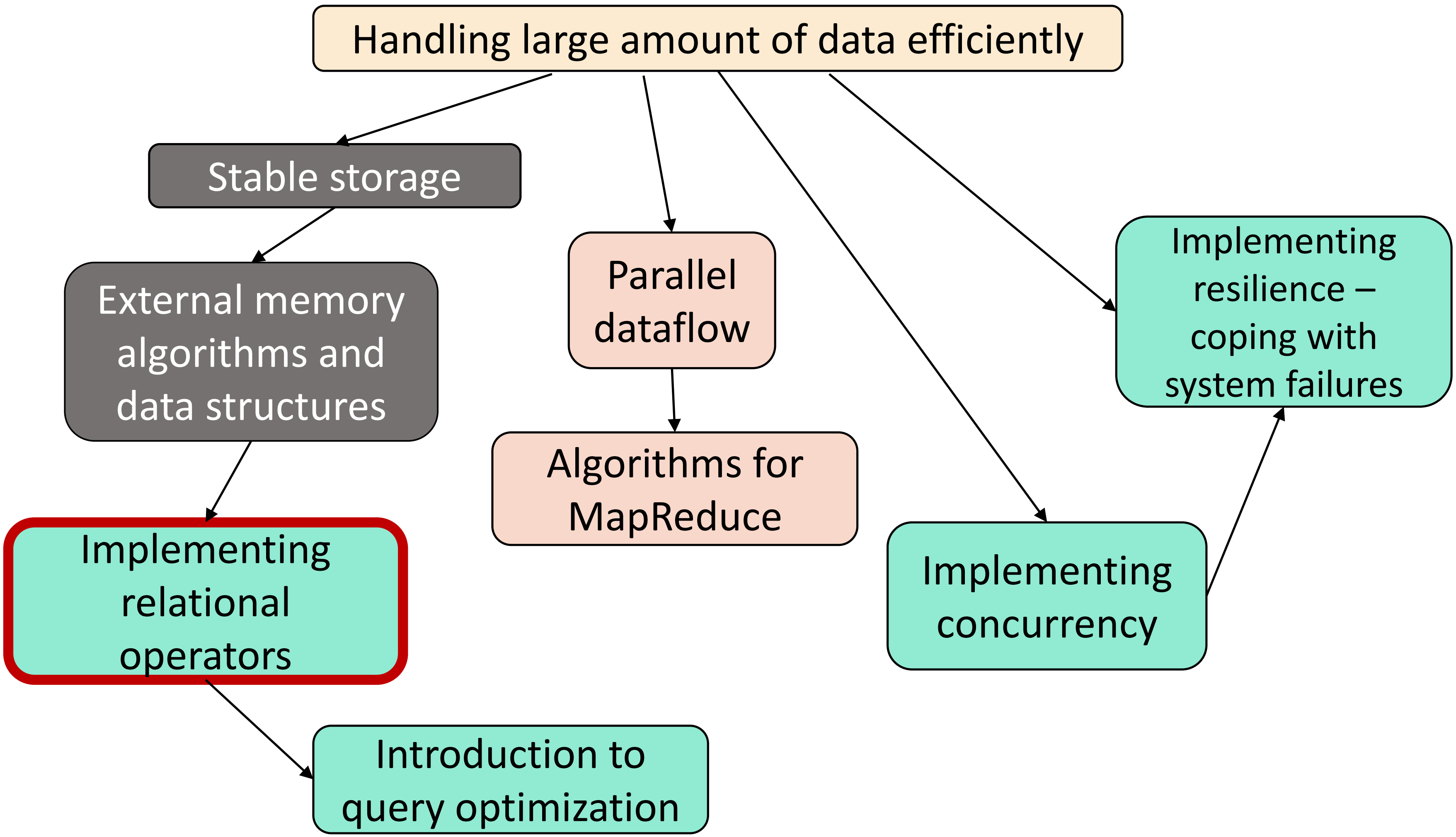
Roadmap


Handling large amount of data efficiently
Stable storage
External memory algorithms and data structures
Parallel dataflow
Algorithms for MapReduce
Implementing resilience – coping with system failures
Implementing concurrency
Implementing relational operators
Introduction to query optimization

Lecture 03.01.

Algorithms for query evaluation

Selection, Projection

By Marina Barsky
Winter 2017, University of Toronto

Queries about data

Two levels:

1. High-level: formulating queries about data (in SQL?)
2. Low-level: **implementing algorithms for answering (*evaluating*)** these queries

Query evaluation

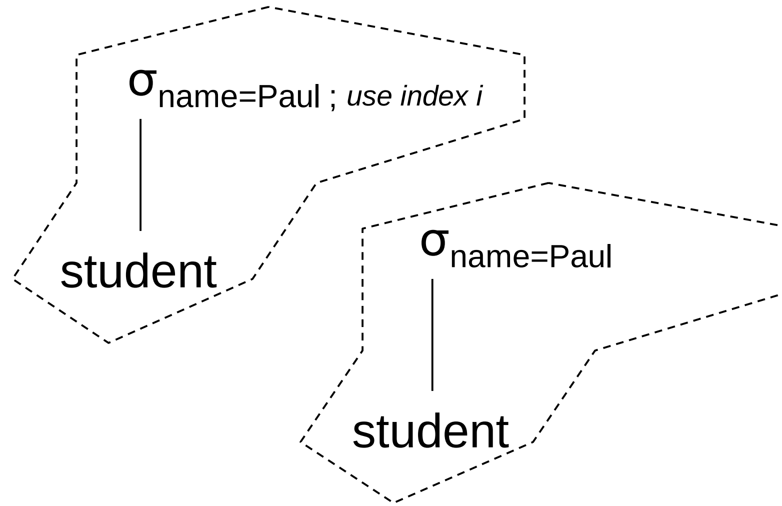
- Efficient implementation for each relational operator
- Combining these implementations into a larger program to answer a given query
- Optimizing query plan before executing this combination

We learn how to implement:

- **Selection** (σ): select a subset of rows from relation
- **Projection** (π): delete unwanted columns from relation
- **Join** (⋈): combine two relations according to a given criteria

Questions to answer:

- What options 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

$\sigma_{name=Paul}$(student)

- scan file
- use secondary index on student.name

Estimating cost

- We use **the number of disk I/Os** measured **in units of 1 block**
- We assume that the **input** for each operator is on disk, but we exclude the cost of writing an **output**:
  - The cost of writing the output to disk *depends on the size of the result, **not on the way the result was computed***
  - We can often ***pipeline*** the result to other operators in main memory

Cost parameters

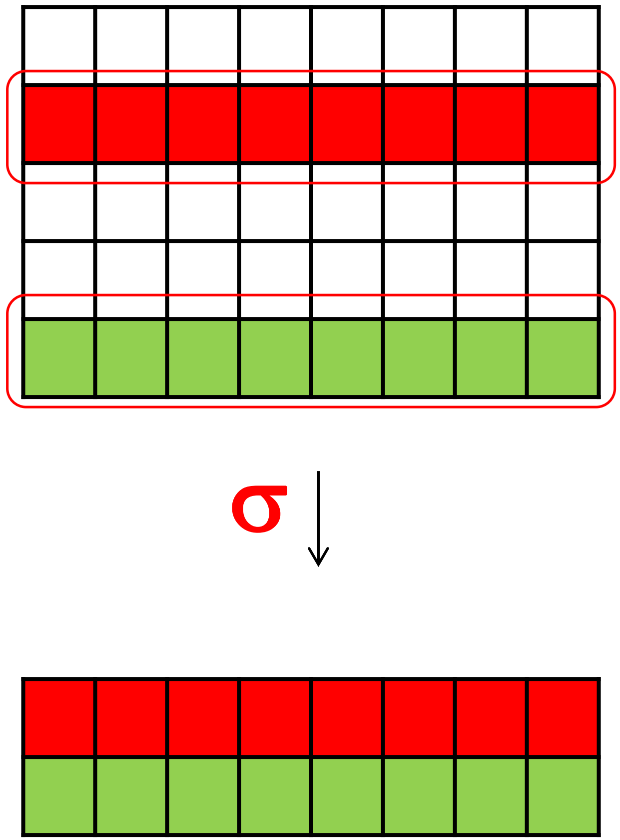
- **R**: the name of the relation on disk
- **M**: number of main memory buffers available (1buffer = 1block)
- **B(R)**: number of blocks in R
- **T(R)**: number of tuples in R
- **V(R, a)**: number of distinct values in column ***a*** of R
- **V(R, L)**: number of tuples in R that differ by at least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

We also need:

R: the name of the relation
M: number of main memory pages
B: number of blocks in R
T: number of tuples in R
V(R, a): cardinality of 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)= T(R)/ V(R,a)* (uniform distribution assumption)
- **HT_i**: number of levels in index *i* (for example, height of B-tree)

Slice operator: Selection



$$S=\sigma_{condition}(R)$$

Select operator: corresponds to WHERE clause

R

A	B	C	D
x	x	1	7
x	y	5	7
y	x	12	3
y	y	23	10

$\sigma_{A=B \wedge D > 5}(R)$

A	B	C	D
x	x	1	7
y	y	23	10

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

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 algorithm II:
R is sorted on selection condition

- 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: R is sorted on selection condition

R

Sorted on <A,C>

A	B	C	D
x	x	1	7
x	y	5	7
x	x	12	3
y	y	23	10
y	x	32	9

$\sigma_{A=y \wedge C > 12}(R)$

A	B	C	D
y	y	23	10
y	x	32	9

```
SELECT *
FROM R
WHERE A = y AND C > 12
```

Selection II: cost

R: the name of the relation
M: number of main memory pages
B: number of blocks in R
T: number of tuples in R
V(R, a): cardinality of column ***a*** of R

- 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: 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

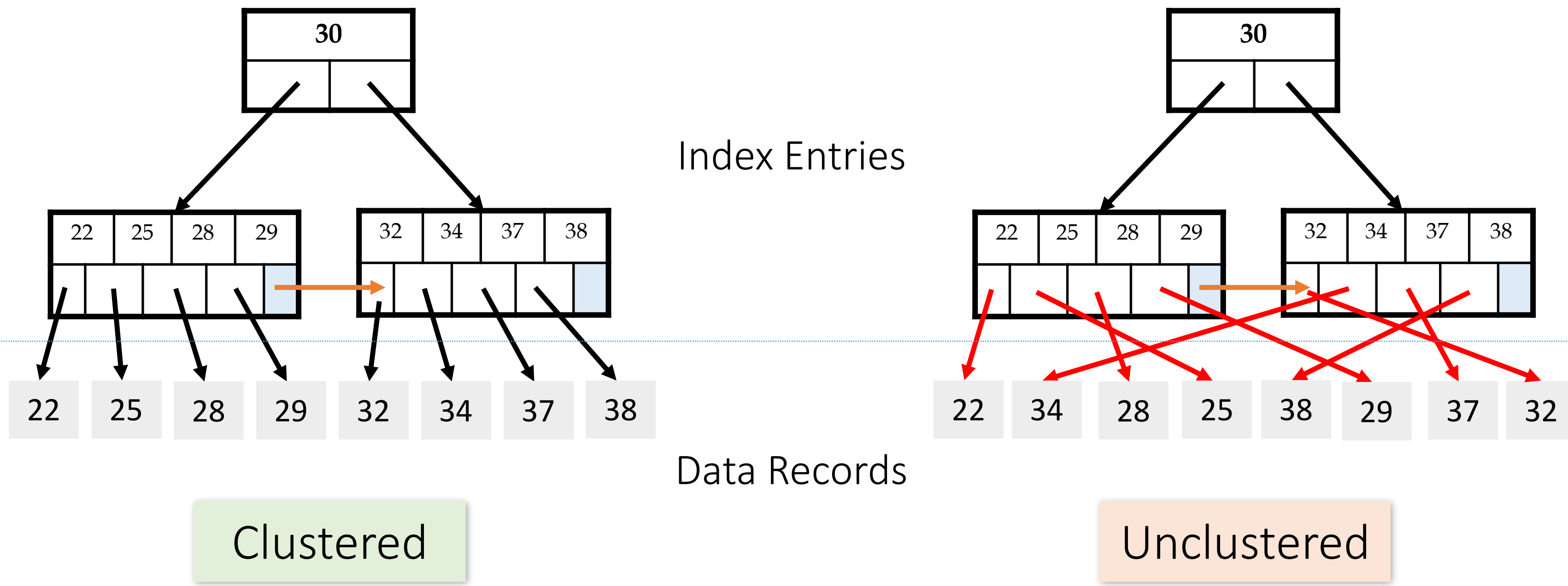
Selection III: cost

- 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 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 could 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*
  → 1000 disk I/O's
- *R* has a clustered index on *a*, V(R,a) = 100.
  → 1 + 1000/100 = 11 I/O's
- *R* has a non-clustered index on *a*, V(R,a) = 100
  → 1 + 20,000/100 = 201 disk I/O's.
  If V(R,a) = 20,000 (*i.e.* attribute *a* is key)
  → 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

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

Hard task to keep R sorted

$\log_2 B(R) + B(R)/V(R,a)$

- Clustered index on R: index search + sequential scan

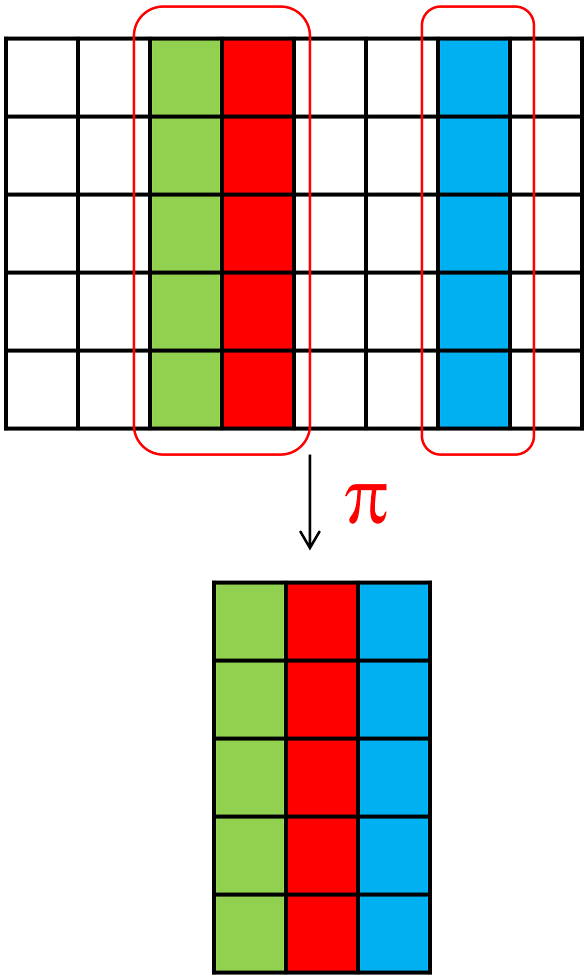
$HT_i + B(R)/V(R,a)$

- Unclustered index on R: index search + non-sequential retrieval

$HT_i + T(R)/V(R,a)$

- Space requirements: **$M \geq 1$** block

Slice operations: Projection



$S=\pi_{\text{attribute list}}(R)$

Projection operator: bag or set?

R

A	B	C	D
x	x	1	7
x	y	5	7
y	x	12	3
y	y	23	10

Bag projection –
in practice

A	D
x	7
x	7
y	3
y	10

SELECT A, D
FROM R

Set projection –
in RA theory

$\pi_{A,D}(R)$

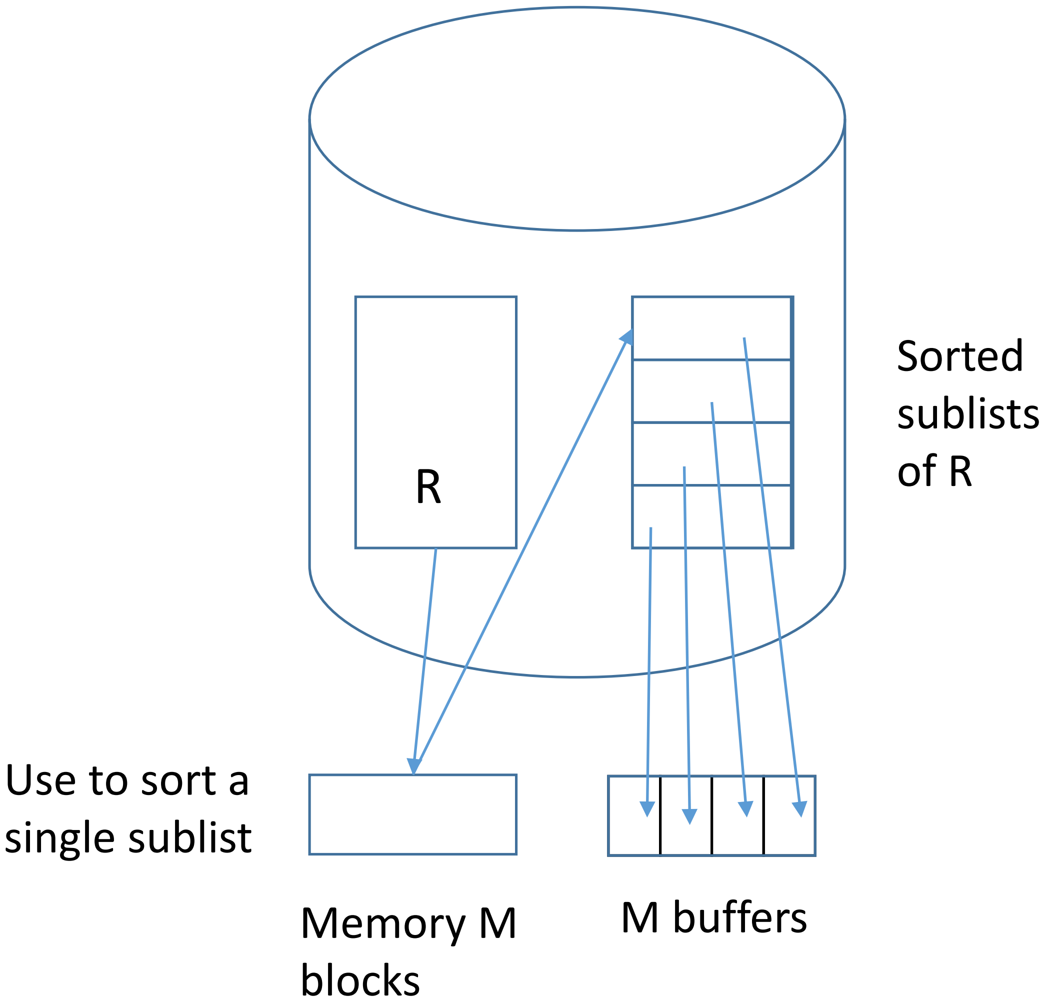
A	D
x	7
y	3
y	10

SELECT DISTINCT A, D
FROM R

Set projection algorithm I: modified 2PMMS

- Sort using a1,a2… 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 I: diagram



Set projection 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 output is not included)
- The total cost of sorting-based projection: **3B(R)** disk I/Os

Set projection I: memory requirements

- Assuming M blocks of memory are available, we create sorted runs of size ~M each
- For the second phase, we need 1 block for each run in main memory to a maximum of ~M blocks
- Thus, $B < M^2$, and the memory requirement is

M >= sqrt(B)

Projection algorithm II: hashing

Phase I: partitioning

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

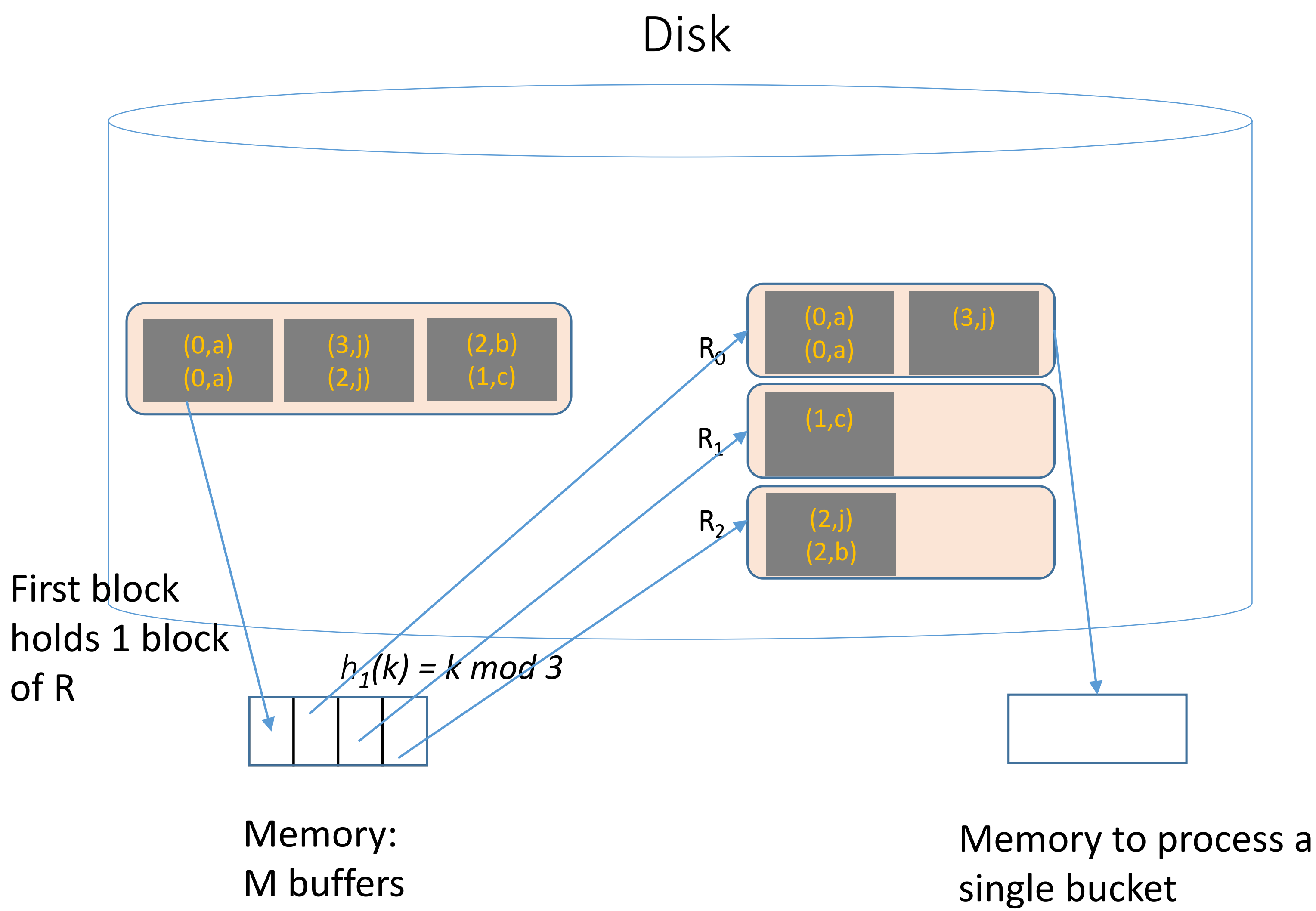
Projection algorithm II: hashing

Phase II: duplicate elimination

- 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 II: diagram



Projection 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 >= sqrt (B)**

Projection III: using indexes

- If an index 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
  - the result is sorted.

Quick question

What implementation would have a smaller cost – implementation for bag projection or set projection? Why?

A. Set projection. The number of records in a set is typically smaller than a bag, and cost is a function of the number of records in the collection.

B. Bag projection. A bag is easier to reason about formally, and therefore allows more aggressive optimization opportunities.

C. Bag projection. Removing duplicates requires an extra step, which can be expensive and is not always required by the application.

Quick question

What implementation would have a smaller cost – implementation for bag projection or set projection? Why?

A. Set projection. The number of records in a set is typically smaller than a bag, and cost is a function of the number of records in the collection.

B. Bag projection. A bag is easier to reason about formally, and therefore allows more aggressive optimization opportunities.

C. Bag projection. Removing duplicates requires an extra step, which can be expensive and is not always required by the application.

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
  - For a complex query involving a chain of operators this gives major savings in I/Os
  - The operators communicate through the ***Iterator*** interface

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

Iterators

- Operators are often implemented as ***Iterators***, which allows to 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

Examples of Iterators

The following pseudocode is given to help you with A1.3

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)

```
Open () {
}
GetNext () {
    t: = input.GetNext()
    next: = NotFound
    if (t != NotFound) {
        if (t satisfies selection condition)
            next: = t
    }
    return next
}
Close () {}
```

Iterator for Projection II (hashing) Takes as an input table-scan or selection GetNext()

Part I: partitioning R into M-1 buckets

```
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
                 …
```

Iterator for Projection II (contd.)

***Open* ()** {

…

initialize 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

}

Part II: setup first bucket

Note: All the preparatory work is done in ***Open***, so we can produce tuple-at-a-time when asked for ***GetNext***

Iterator for Projection II: GetNext

```
GetNext () {
    next: = NotFound
    if (t is past the last tuple on block b of R_i) {
        increment b to the next block;
        if (there is no next block) {
            increment i to the next bucket i+1
            if (there is no next bucket)
                return NotFound

            empty in-memory hash table
            b: = first block of R_i
            t: = the first tuple of b
    }
    try to insert t into in-memory hash table
    while (collision and t is in hash table) {
        t: = GetNext ()
        if ( t=NotFound )
            return NotFound
    }
    next: = t
    increment t to the next tuple of b
    return next
```

Processes current tuple of current bucket Ri

Tries current tuple for duplicates