IN3020&4020 – Database Systems (2024)
Part 1: Query Processing
Lectures 13, 14, 15: Query Evaluation

---

27 February, 4, 5 March

Egor V. Kostylev
IFI, University of Oslo
egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases

Part 2. Transaction management in relational databases

Part 3. NoSQL DBMS

SQL query processing relies on two key ingredients:
Relational Algebra (✓) and Indexes (✓)

We have studied these ingredients (almost in isolation)
and now put all things together

SQL (DQL) query processor in action

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

$\downarrow$ parsing

(parse tree)

$\downarrow$ compilation

$$\pi_{title}(\sigma_{starName=name}(\mathbf{StarIn} \times \pi_{name}(\sigma_{birthDate\ LIKE\ `\%1960'}(\mathbf{MovieStar}))))$$

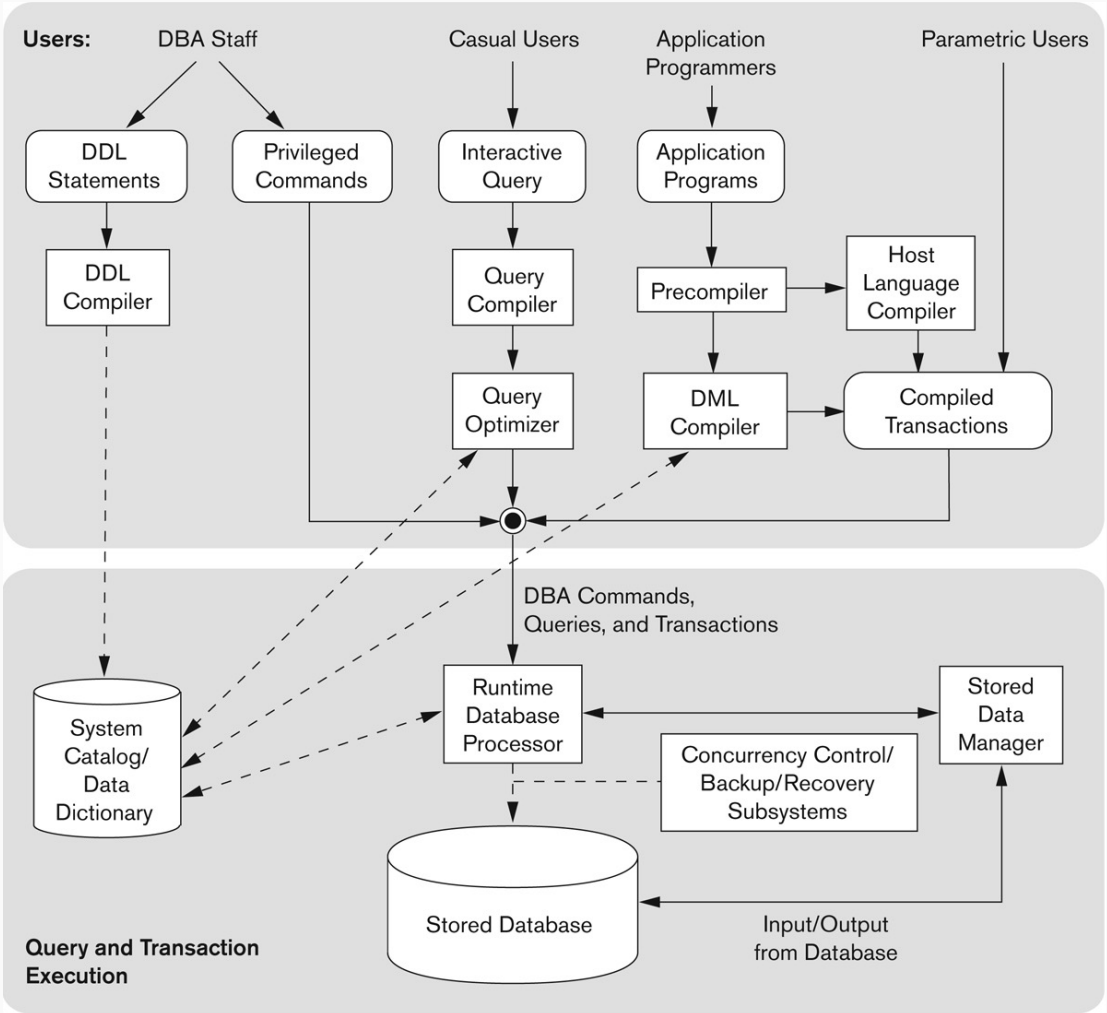
$\downarrow$ optimisation

$$\pi_{title}(\mathbf{StarIn} \bowtie_{starName=name} \pi_{name}(\sigma_{birthDate\ LIKE\ `\%1960'}(\mathbf{MovieStar})))$$

$\downarrow$ evaluation over tables

(Result)

The place of parsing and optimisation



Materials to read about query compilation and optimisation

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapter 18
   (my lectures have different accents)

2. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Section 15
   (much closer to my presentation)

3. etc.

Lecture Plan

1. Overview
2. Executing Basic Operations
3. Executing RA Operations

3.1. Unary Operations

3.2. Binary Operations

4. Summary
5. Query Plans in Real DBMSs

1. Overview

Physical Plans

Logical query plan: RA expression

Physical query plan: Logical query plan where an algorithm is assigned to each operation

In principle, we may have a choice between several algorithms for an operation with different incomparable characteristics (e.g., time consumption) and requirements (e.g., necessary memory, presence of indexes, etc.)

Moreover, algorithms may be pipelined so that characteristics of two algorithms together may be better than the sum of their characteristics in isolation

We used the size of the intermediate relations as costs of logical query plans, which are an approximation of time consumption of the algorithms

More fine-grained costs take the concrete algorithms into account, and estimate different physical plans rather than just logical plans

Algorithms for operations

For simplicity, in what follows we ignore this issue and assume that we have a logical plan constructed

We will only study algorithms 'in isolation', sometimes several for one operation (depends on available memory, indexes, etc.)

Before algorithms for the RA operations, we will consider algorithms for basic operations (sorting, hashing, etc.)

- need them by themselves, e.g., as intermediate steps
- they are often the base for RA algorithms

Algorithms for operations: Assumptions and costs

Our (new) main cost: number of block I/O (reads and writes)

For each algorithm, we generally assume

- the operands are initially on the disk
- the result is written to output stream (with no cost)
- other costs can be ignored

- the tuples of a relation are clustered—that is, tightly packed in blocks (no free space in a block for a new tuple, maybe spanned or unspanned)

- when we use indexes, we ignore the cost of their retrieval

Notation

We use the following parameters:

$MMBuffers$ – the size of the main memory, in blocks
(sometimes called buffers)

$Blocks(R)$ – the number of blocks to store relation R on disk

$Tups(R)$ – the number of tuples in relation R

$Vals(R.A)$ – the number of different values of (one or several) attributes A in R

($Tups(R)$ and $Vals(R.A)$ are now exact, not estimation)

2. Executing basic operations

Scanning a relation

Scanning a relation is the most simple: just to retrieve the whole relation

- Table scan: no index used, blocks retrieved one by one

  Cost: $Blocks(R)$ disk I/O operations

  (If clustered; otherwise, up to $Tups(R)$, or even more)

- Index scan: Index (sparse or dense) is used

  Cost: $Blocks(R)$ disk I/O operations (the same)

  (But may be used to retrieve a part of R quicker)

Sorting a relation

Sort scan reads a relation R and returns it in sorted order with respect to a set of attributes *Atts*

- if R has an index of *Atts*, then it can be used to traverse the tuples in R in the order
  Cost: $Blocks(R)$ (primary/clustering) or
  $Tups(R)$ (secondary) disk I/O operations

- if $Blocks(R) \leq MMBuffers$ (i.e., the whole R fits into the main memory), we use an efficient sorting algorithm
  Cost: $Blocks(R)$ disk I/O operations

- if $Blocks(R) > MMBuffers$ (i.e., R is too big to fit into memory), we use a dedicated sorting algorithm that handles data in multiple passes
  we consider two-phase multiway merge sort (TPMMS)
  (has requirement $\sqrt{Blocks(R)} \leq MMBuffers$,
  but can be generalised to N-phase multiway merge sort)

Two-Phase Multiway Merge Sort (TPMMS): Phase 1

TPMMS Phase 1. Sort parts of R of size *MMBuffers* individually:

Repeat:

1. Fill all (*MMBuffers*) buffers with new blocks of R
2. Sort this part in the memory
3. Write the sorted result back to disk

Until all R is processed

Cost of Phase 1: $2 * Blocks(R)$

Two-Phase Multiway Merge Sort (TPMMS): Phase 2

TPMMS Phase 2. Merge all the sorted parts into one:

Repeat:

1. Read the first block from each part into buffers (at most *MMBuffers*)
2. Find the smallest tuple over the first tuples in the buffers
3. Remove this tuple from the buffer; if the buffer is empty, re-fill it with the next block from the same part
4. Put this tuple to the output buffer; if it is full, flush it

Until all parts in the main memory is empty

Cost of Phase 2: $Blocks(R)$

Total cost: $3 * Blocks(R)$

Requirement: Number of parts cannot be more than $MMBuffers - 1$:

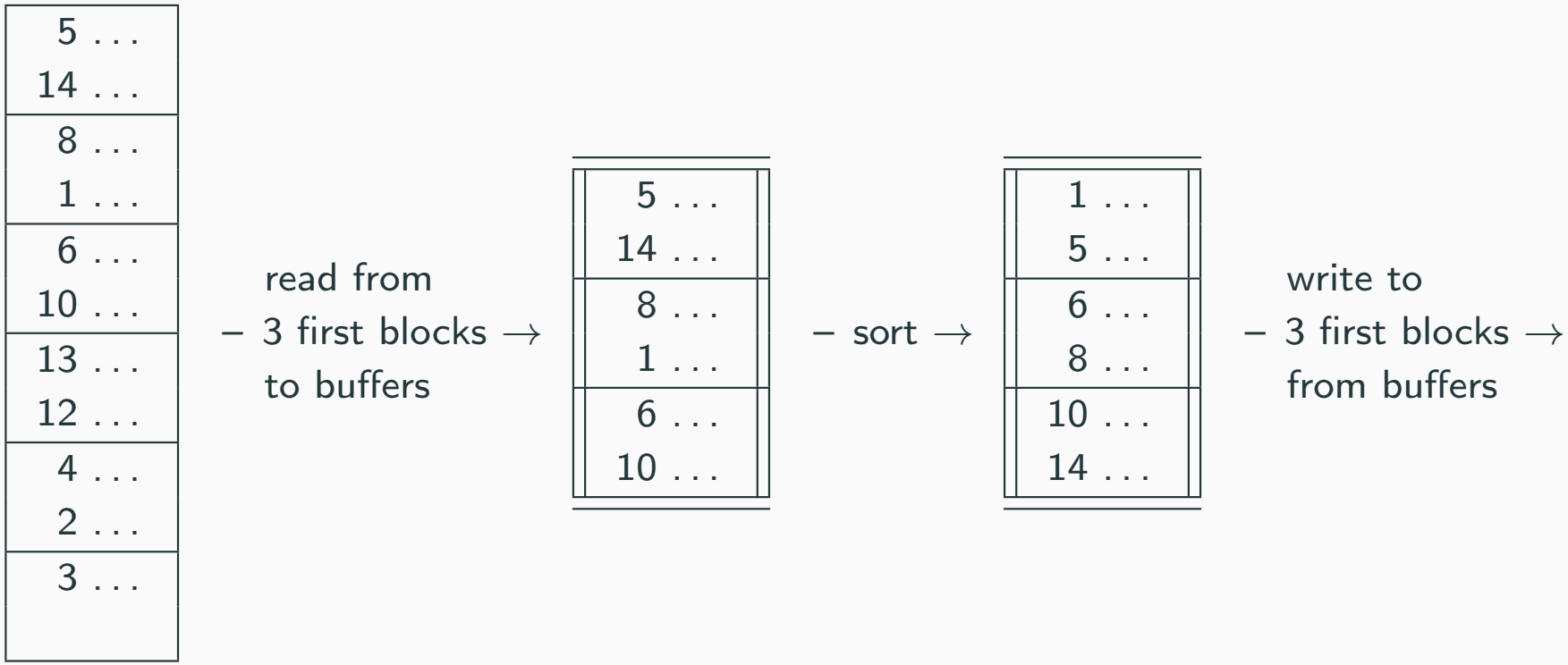
$Blocks(R)/MMBuffers \leq MMBuffers - 1$

so essentially $\sqrt{Blocks(R)} \leq MMBuffers$

TPMMS: Example Phase 1

Let $MMBuffers = 3$

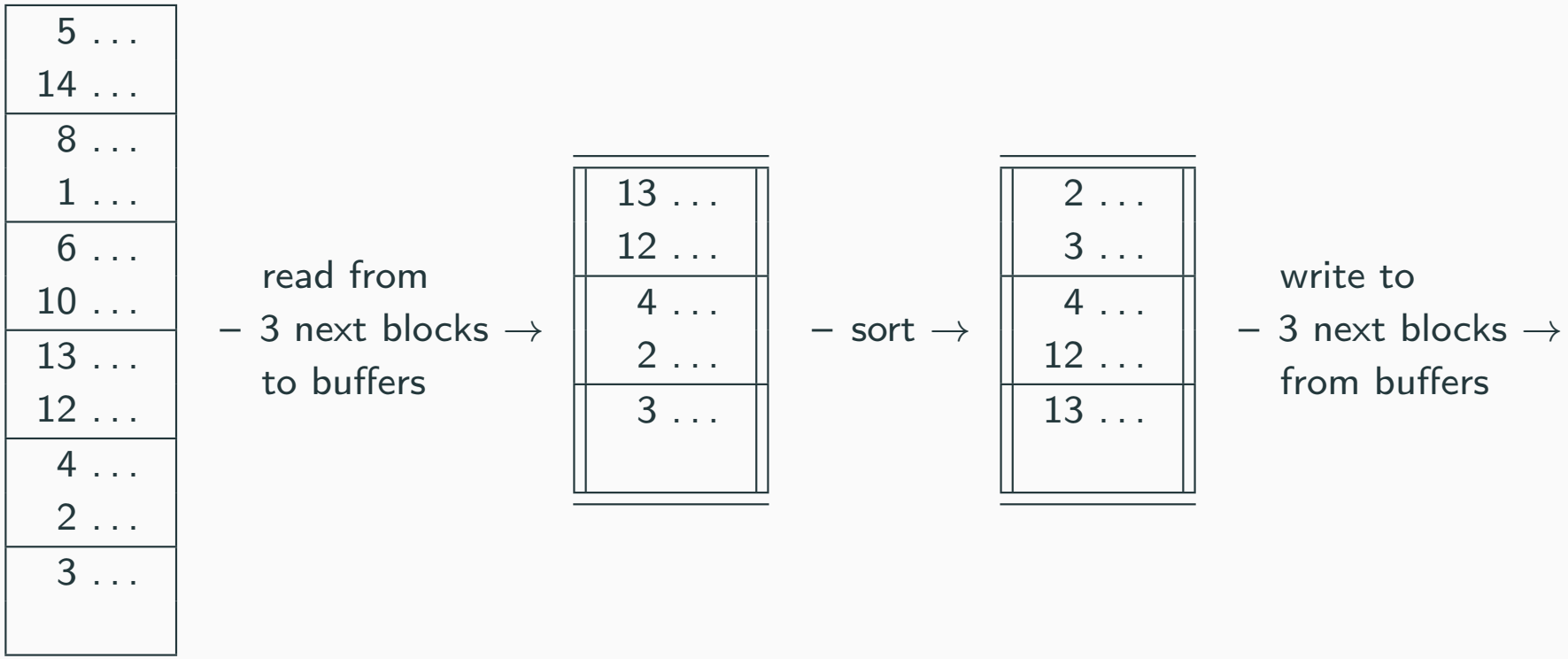
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 1

Let $MMBuffers = 3$

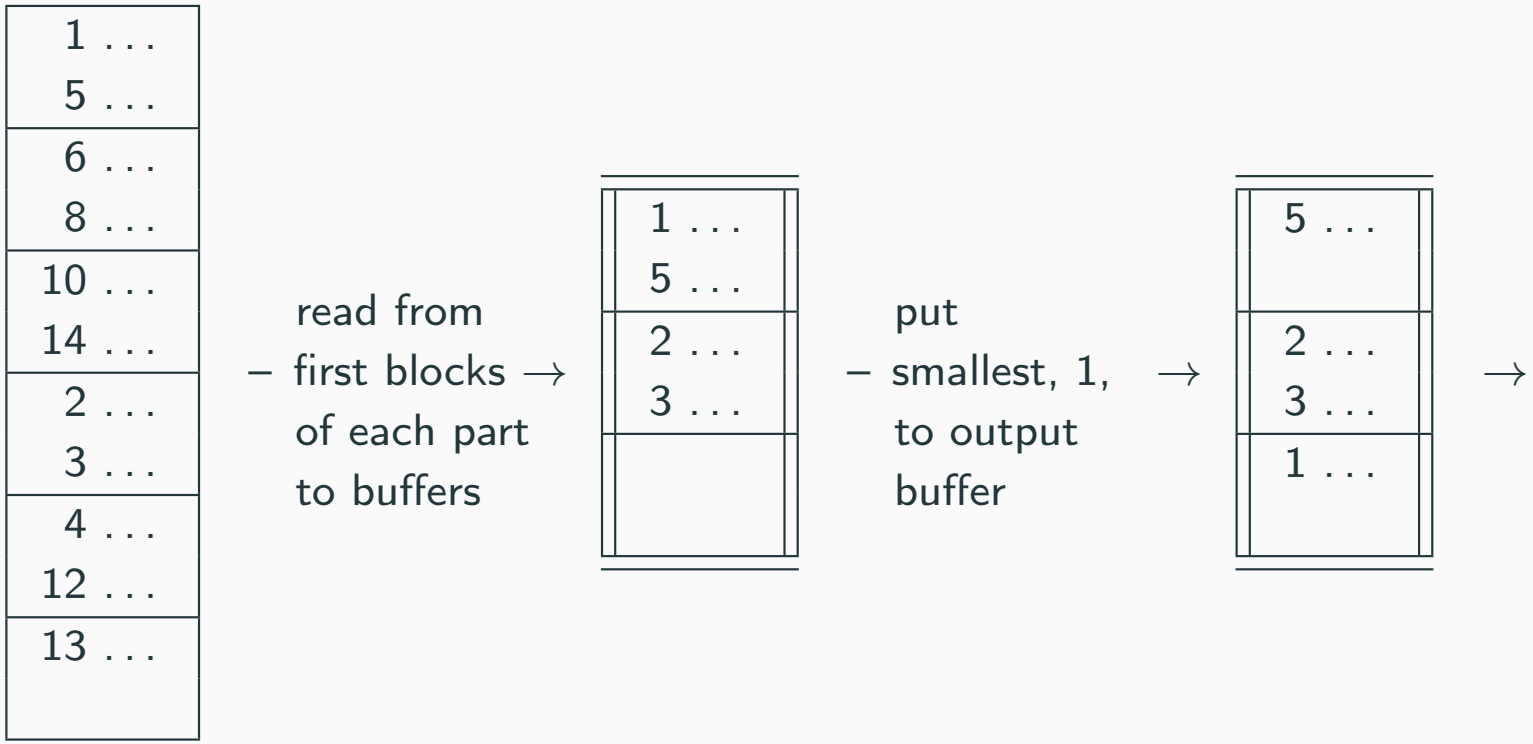
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

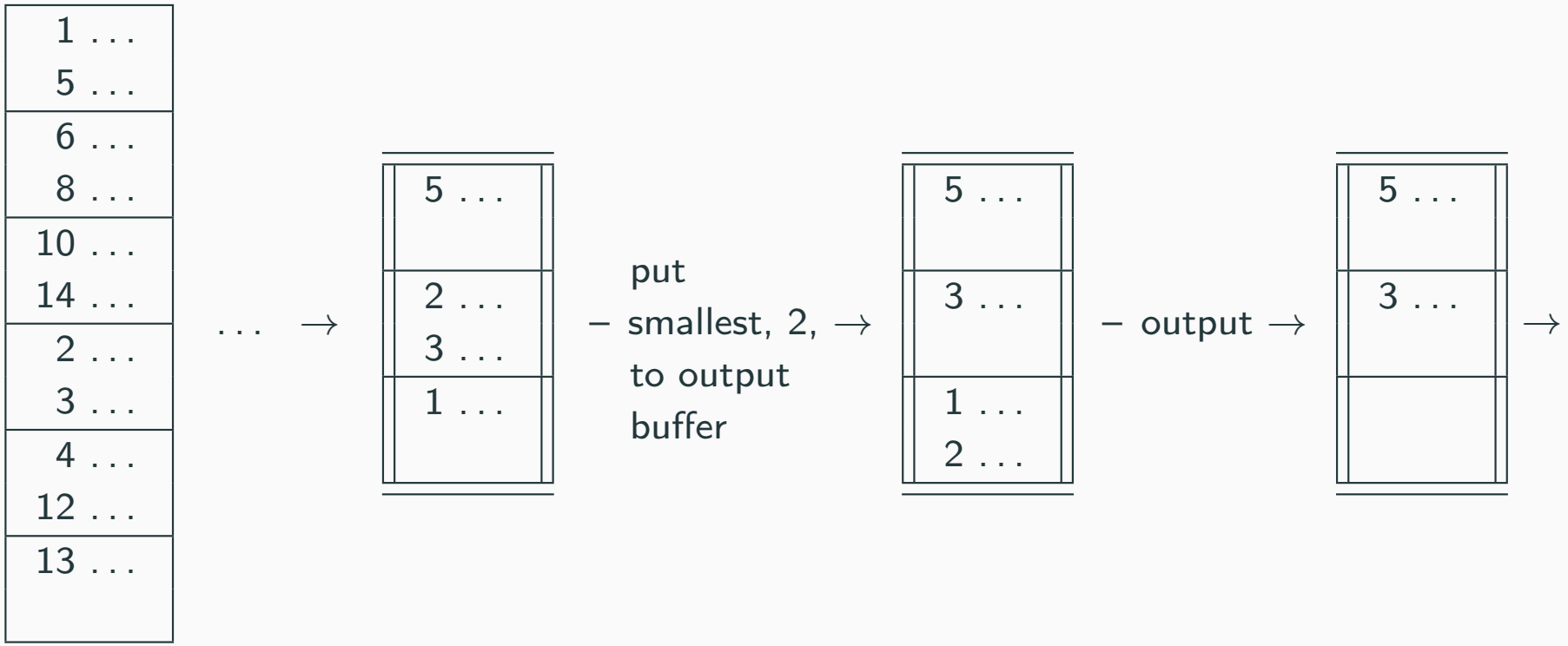
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

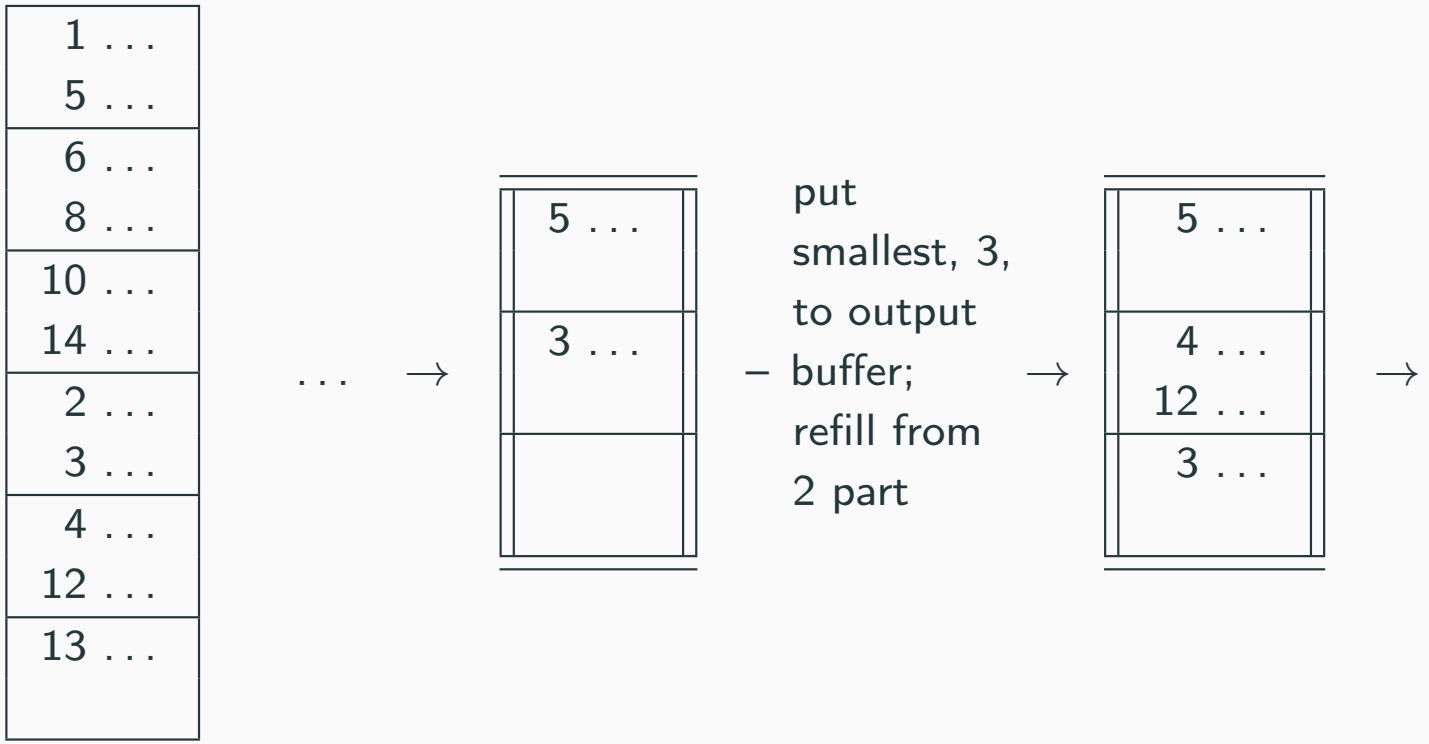
Consider $Tups = 11$ tuples in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

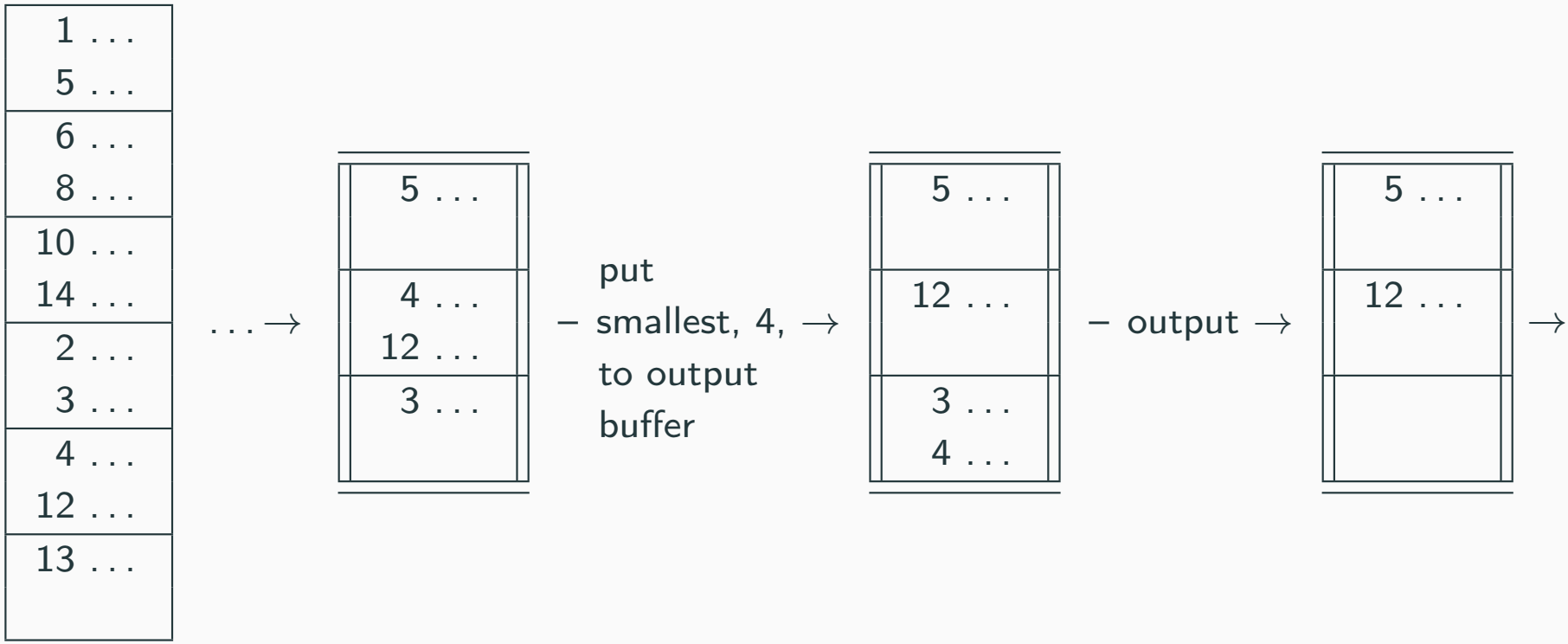
Consider $Tups = 11$ tuples in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

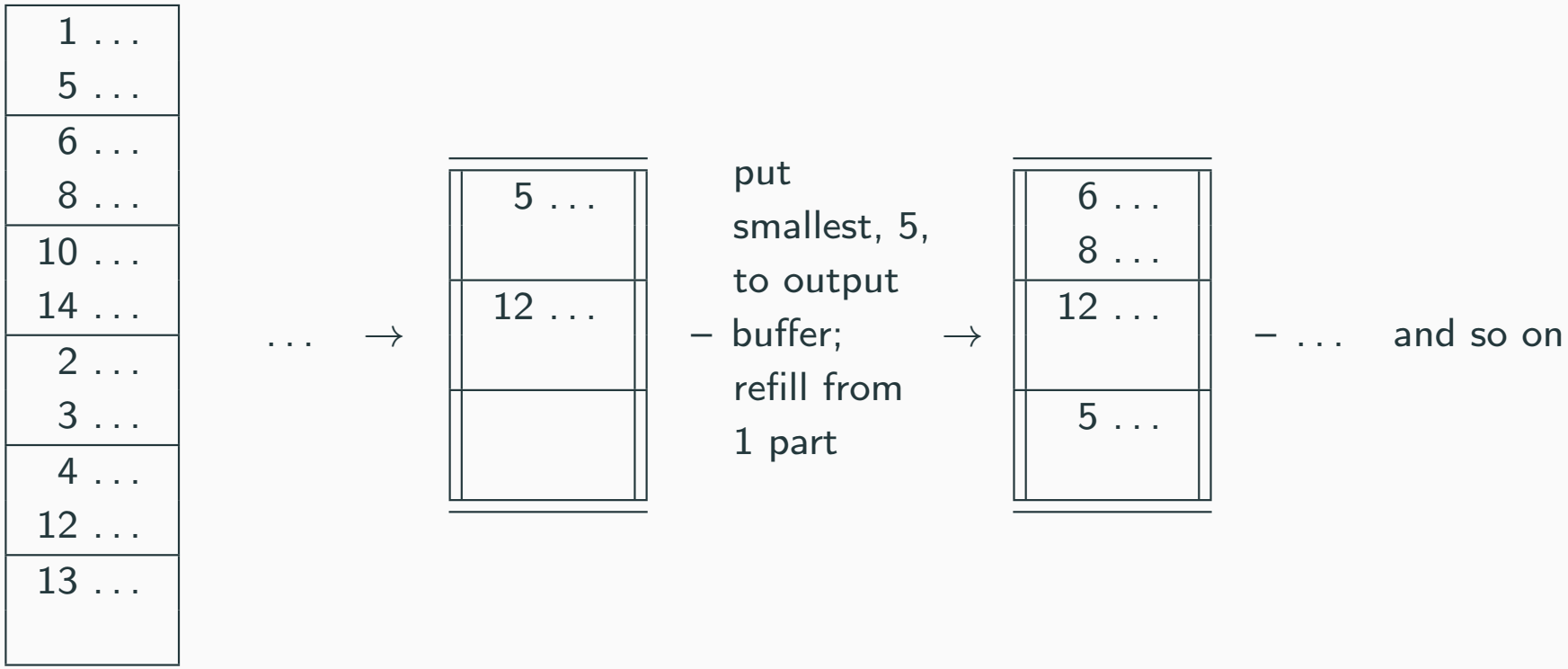
Consider $Tups = 11$ tuples in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

Consider $Tups = 11$ tuples in $Blocks = 6$ (2 tuples per block):



Hash Partitioning

Partitions relation R to buckets according to hash function h
(applied to one or more attributes of R)

Condition: the number of hash values (i.e., buckets) is no more than $MMBuffers - 1$

Uses one read buffer and (at most) $MMBuffers - 1$ buffers associated to the hash values

Repeat:

1. If the read buffer is empty,
   load the next block of R to the read buffer
2. For each tuple t in the read buffer
   - add t to the bucket buffer $h(t)$
   - if the bucket buffer is full, flush it to the output

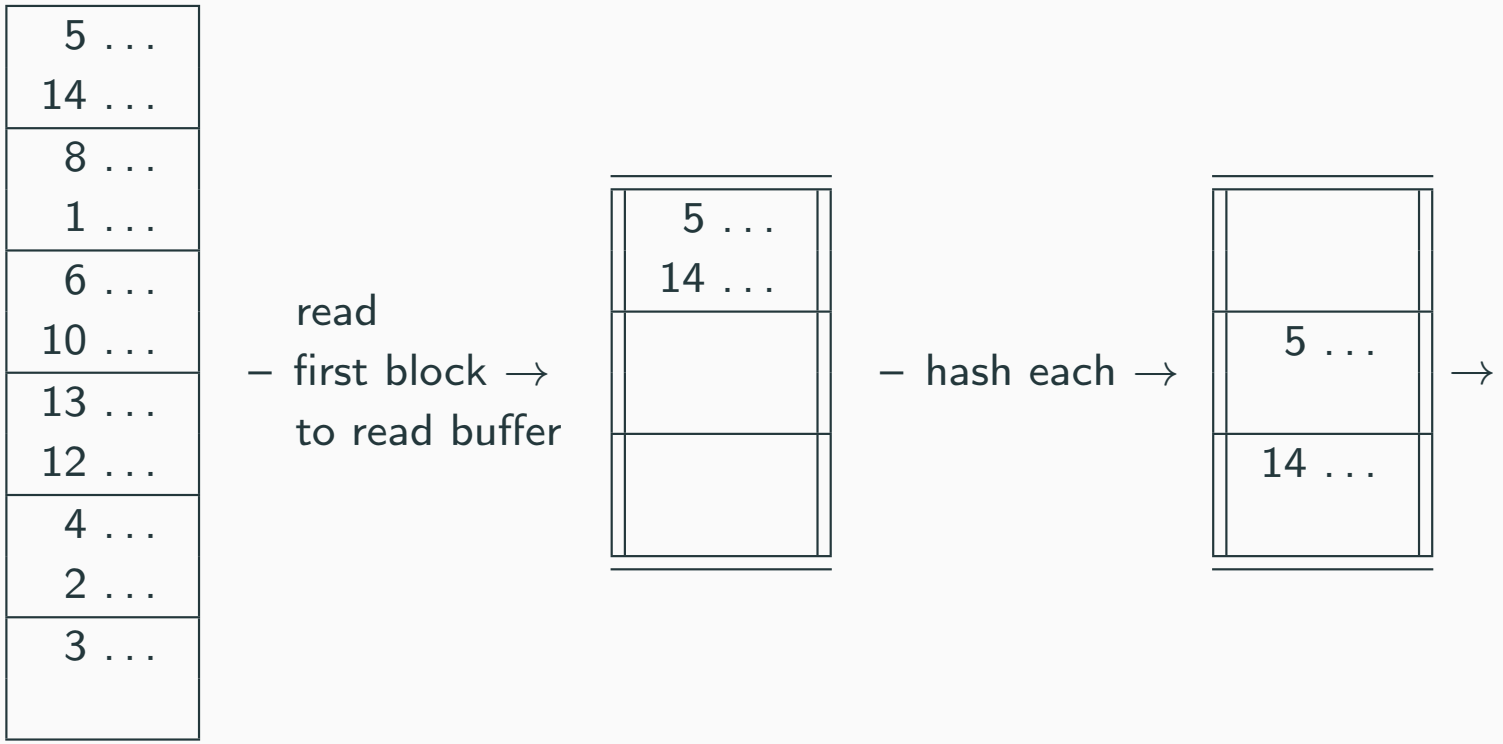
Until all blocks of R are processed

Cost: $Blocks(R)$ (or up to $Tups(R)$ if not clustered)

Hash Partitioning: Example

Let $MMBuffers = 3$ and $h(t)$ is $Mod\ 2$

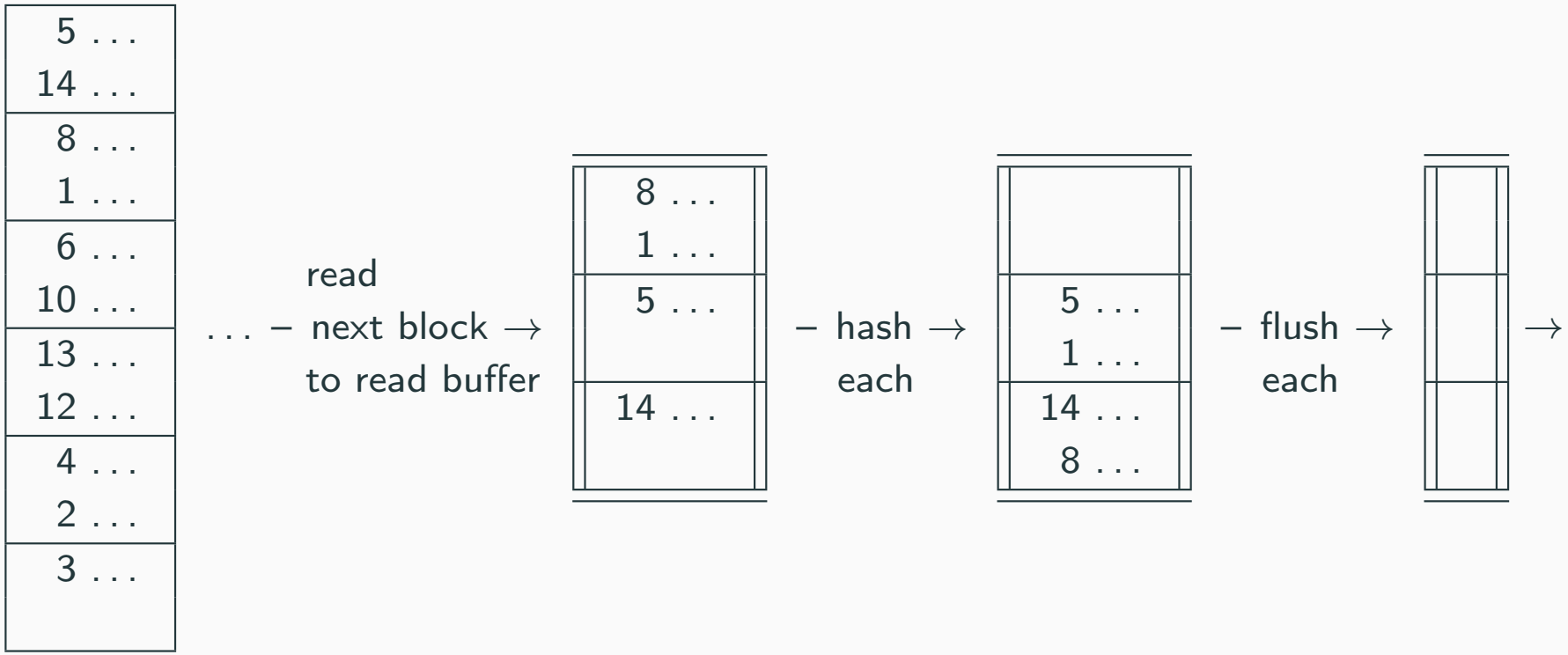
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



Hash Partitioning: Example

Let $MMBuffers = 3$ and $h(t)$ is $Mod\ 2$

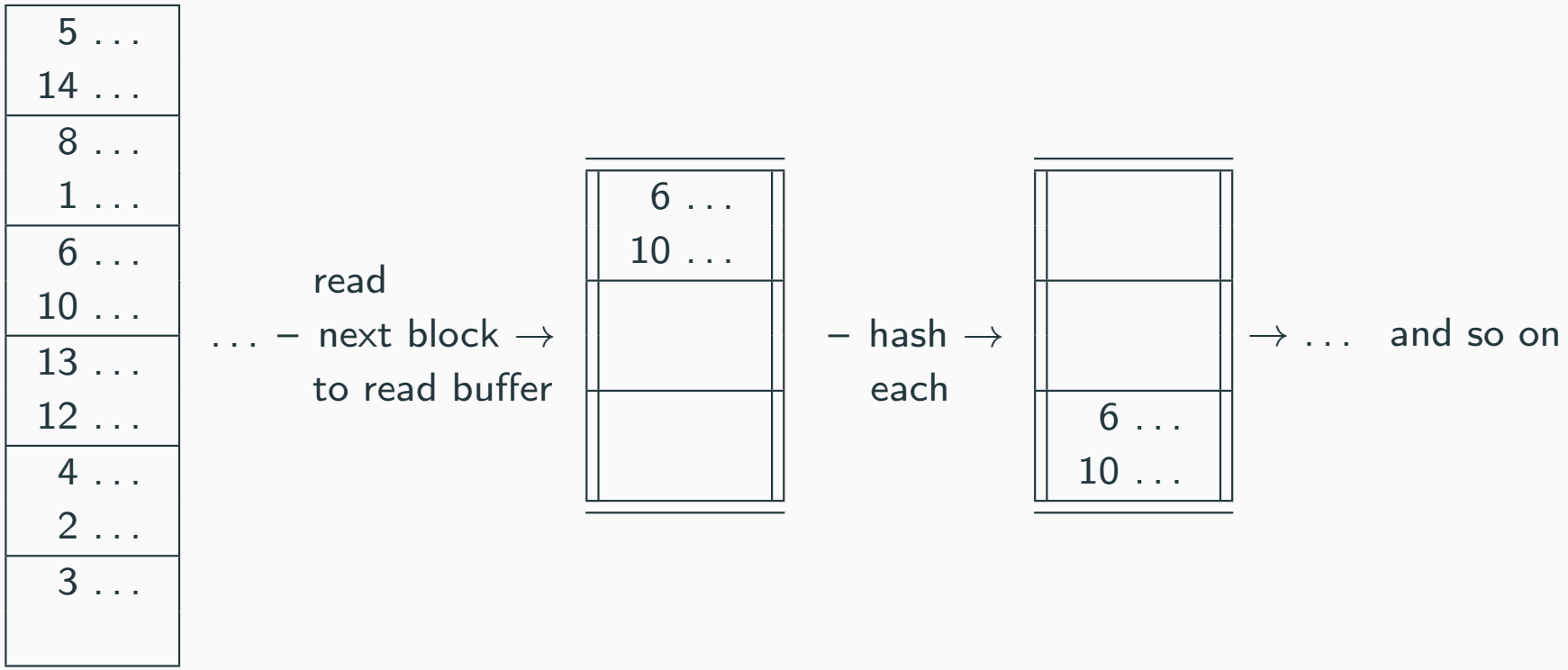
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



Hash Partitioning: Example

Let $MMBuffers = 3$ and $h(t)$ is $Mod\ 2$

Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



3. Executing RA Operations

Algorithms: Overview

Classes based on approach:

- Sort-based (always work)
- Hash-based (relies on a good hash function)
- Index-based (relies on indexes)

Classes based on memory requirements:

- One-pass: data fits into the main memory,
  read once
- Two-pass: data does not fit into the memory,
  but two passes are enough
- N-pass, $N \geq 3$: recursive generalisation to more memory,
  several passes (mostly omitted below)

3.1. Unary operations

Tuple-at-a-time operations: Selection and Projection

Selection $\sigma_{Cond}(R)$ and projection $\pi_L(R)$ process each tuple of R in isolation, so they have simple algorithms, regardless of $Blocks(R)$:

Repeat:

1. Retrieve the next relevant block into a buffer
   (use indexes if available and relevant)
2. Process tuples in the buffer one by one, output when relevant

Until all relevant blocks of R are processed

Cost (I/O reads):

- $Blocks(R)$ for selection without index and projection
- for selection with index, depends on condition and index type
  for example, $\lceil Blocks(R)/Vals(R.A) \rceil$ on average
  for $A = c$ as $Cond$ and primary/clastering index

Requirement: (just) $1 \leq MMBuffers$

Selection with secondary index: Example

Let $Tups(R) = 20000$, $Blocks(R) = 1000$

Cost of $\sigma_{A=c}$ with index on A:

- for $Vals(R.A) = 10$ we have $1000/10 = 100$ disk I/O
- for $Vals(R.A) = 100$ we have $1000/100 = 10$ disk I/O
- for $Vals(R.A) = 20000$ (i.e., candidate key) we have $\lceil 1000/20000 \rceil = 1$ disk I/O

One-pass duplicate elimination

Duplicate elimination $\delta(R)$

when it can be performed in main memory

Use one read buffer and other buffers to store the result

Repeat:

1. Retrieve the next block of R in the read buffer
2. For each tuple t in the read buffer,
   search for the same tuple in the result buffers,
   append if not found

Until all blocks of R are processed

Cost (I/O reads): $Blocks(R)$

Requirement: $Blocks(\delta(R)) \leq MMBuffers - 1$

Two-pass sort-based duplicate elimination

Duplicate elimination $\delta(R)$

when it cannot be performed in the main memory

Sort-based algorithm: essentially TPMMS where duplicates are eliminated before output

Phase 1: Sort parts of size *MMBuffers* individually with all attributes as the sort key

Phase 2: Merge the parts eliminating duplicate tuples when output

Cost: $3 * Blocks(R)$

Requirement: $\sqrt{Blocks(R)} \leq MMBuffers$

Two-pass hash-based duplicate elimination

Duplicate elimination $\delta(R)$

when it cannot be performed in the main memory

Hash-based algorithm: exploit that duplicates will have the same hash value

Partition the relation into buckets, write buckets back to disk

Run one-pass duplicate elimination individually on each bucket

Cost: $3 * Blocks(R)$

Requirement: The size of each bucket ($MMBuffers - 1$ in total) can't be more than $MMBuffers$:

$Blocks(R) \leq MMBuffers * (MMBuffers - 1)$

so essentially $\sqrt{Blocks(R)} \leq MMBuffers$

(Assuming equal distribution over buckets, i.e., good hash function)

The same bound as sort-based, but quite different approach

Grouping with Aggregation

Grouping and Aggregation γ_L is very similar to duplicate elimination δ

One-pass: same idea, keep a tuple for each group in memory
Cost: $Blocks(R)$
Requirement: $Blocks(\gamma_L(R)) \leq MMBuffers - 1$ (little more for `AVG`)

Sort-based two-pass: same idea, except that the sort key is the grouping attributes
Cost : $3 * Blocks(R)$
Requirement: $\sqrt{Blocks(R)} \leq MMBuffers$ (approximately)

Hash-based two-pass: same idea, except that the hashed key is the grouping attributes
Cost : $3 * Blocks(R)$
Requirement: $\sqrt{Blocks(R)} \leq MMBuffers$
(assuming good hash function)

3.2 Binary operations

One-pass bag union

Bag union $R \cup^b S$ is essentially tuple-at-a-time

1. Retrieve and output all blocks of R one by one
2. Retrieve and output all blocks of S one by one

Cost: $Blocks(R) + Blocks(S)$

Requirement: $MMBuffers \geq 1$

One-pass set union

Set union $R \cup S$: need to eliminate duplicates

Keep the smaller of R and S in $MMBuffers - 1$ buffers, use the other buffer to pass through the bigger one

Assuming $Blocks(R) \leq Blocks(S)$,
load all blocks of $Blocks(R)$ into the $MMBuffers - 1$ buffers,
output all tuples in $Blocks(R)$ blocks (keeping in buffers)
Repeat:

1. Retrieve the next block of S into the one buffer
2. For each tuple t of the S buffer,
   search for t in the R buffers,
   output t if not found

Until all blocks of S are processed

Cost: $Blocks(R) + Blocks(S)$

Requirement: $\min(Blocks(R), Blocks(S)) \leq MMBuffers - 1$

Two-pass sort-based set union

(Set) union $R \cup S$

when it cannot be performed in the main memory

Sort-based algorithm: essentially duplicate elimination by TPMMS for $R \cup S$

Phase 1: Sort parts of $R \cup S$ of size *MMBuffers* individually with all attributes as the sort key

Phase 2: Merge the parts eliminating duplicate tuples when output

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: $\sqrt{Blocks(R) + Blocks(R)} \leq MMBuffers$

Two-pass hash-based set union

(Set) union $R \cup S$

when it cannot be performed in the main memory

Hash-based algorithm: using that duplicates have same hash value

Partition R and S into buckets separately, write buckets to disk

Run one-pass set union for each pair of buckets with same value

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: at most $MMBuffers - 1$ buckets (for both R and S),
$\min(Blocks(R_i), Blocks(S_i)) \leq MMBuffers - 1$ for each bucket i:

essentially $\sqrt{\min(Blocks(R), Blocks(S))} \leq MMBuffers$

(Assuming equal distribution over buckets,

and that one of R, S is smaller than the other in all buckets)

Better than sort-based, but with an essential assumption

Intersection and difference (set and bag)

Same ideas as for set union:

One-pass:

Cost: $Blocks(R) + Blocks(S)$

Requirement: $\min(Blocks(R), Blocks(S)) \leq MMBuffers - 1$

Two-pass sort-based:

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: $\sqrt{Blocks(R) + Blocks(R)} \leq MMBuffers$

Two-pass hash-based:

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: $\sqrt{\min(Blocks(R), Blocks(S))} \leq MMBuffers$ (almost)

(under the assumptions; note also that the bounds are symmetric even for non-commutative difference)

One-pass natural join

Natural join $R \bowtie S$

when the smaller relation fits to the main memory

Keep the smaller of R and S in $MMBuffers - 1$ buffers, use the other buffer to pass through the bigger one:

Assuming $Blocks(R) \leq Blocks(S)$,

load all blocks of $Blocks(R)$ into the $MMBuffers - 1$ buffers

Repeat:

1. Retrieve the next block of S into the one buffer
2. For each tuple t in the S buffer,

   search for the tuples joining with t in the R buffers,

   output all join results

Until all blocks of S are processed

Cost: $Blocks(R) + Blocks(S)$

Requirement: $\min(Blocks(R), Blocks(S)) \leq MMBuffers - 1$

Nested-loop natural join

Natural join $R \bowtie S$

generalisation of one-pass

Apply one-pass join to the parts of the smaller of R and S separately, using $MMBuffers - 1$ buffers:

Assuming $Blocks(R) \leq Blocks(S)$,
Repeat:

1. load the next $MMBuffers - 1$ blocks of $Blocks(R)$
   into the $MMBuffers - 1$ buffers
2. apply one-pass join to these buffers and full S

Until all blocks of R are processed

Cost: $Blocks(R) + Blocks(S) * \lceil Blocks(R)/(MMBuffers - 1) \rceil$

Requirement: $2 \leq MMBuffers$

(essentially, no space requirement, but quadratic)

Two-pass sort-based natural join

Natural join $R \bowtie S$ via two-pass sorting

First sort separately, then join:

> Sort R and S separately on the join attributes by TPMMS
>
> Assign $MMBuffers/2$ buffers to R, same for S
>
> Repeat:
>
> 1. fill empty R buffers with next R blocks, same for S
> 2. output all joined tuples with the smallest sort values,
>
> delete contributing tuples
>
> Until all blocks of R and S are processed

Cost: $5 * (Blocks(R) + Blocks(S))$ (4 passes for TPMMS, 1 for join)

Requirement 1: $\sqrt{\max(Blocks(R), Blocks(S))} \leq MMBuffers$

Requirement 2: at most $MMBuffers/2$ blocks with the same join value in each of R and S

Two-pass hash-based natural join

Natural join $R \bowtie S$ via hashing

Hash-based algorithm: using that duplicates have same hash value

Partition R and S into buckets separately hashing the join values

Run one-pass join for each pair of buckets with same value

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: at most $MMBuffers - 1$ buckets (for both R and S),
$\min(Blocks(R_i), Blocks(S_i)) \leq MMBuffers - 1$ for each bucket i:

essentially $\sqrt{\min(Blocks(R), Blocks(S))} \leq MMBuffers$

(Assuming equal distribution over buckets,
and that one of R, S is smaller than the other in all buckets)

Two-pass hybrid hash-based natural join

Natural join $R \bowtie S$ via hashing with improvement

Improvement: Keep one bucket of smaller relation in main memory,
and join on the fly when hashing the larger relation
(assuming that the number of buckets allow this)

Cost: $(3 - 2MMBuffers/Blocks(R)) * (Blocks(R) + Blocks(S))$

(Cost of hash-based: $3 * (Blocks(R) + Blocks(S))$)

Homework: understand this advantage

Requirement: essentially
$\sqrt{\min(Blocks(R), Blocks(S))} \leq MMBuffers$

(with same assumptions plus
number of buckets + size of the R bucket $\leq MMBuffers$)

One index based natural join

Natural join $R \bowtie S$ using an index

Iterate over R, get relevant tuples of S using index on join attributes A:

Repeat:

1. read next blocks of R into $MMBuffers - 1$ buffers
2. for each tuple t in the buffers,
   retrieve all tuples of S joining with t using index and last block
   output joined tuples, remove t from the buffers

Until all blocks of R are processed

Cost (estimate) for clustered index (i.e., S is sorted on A):

$$Blocks(R) + Tups(R) * \lceil Blocks(S)/Vals(S.A) \rceil$$

Cost (estimate) for non-clustered index:

$$Blocks(R) + Tups(R) * \lceil Tups(S)/Vals(S.A) \rceil$$

Requirement: $2 \leq MMBuffers$

Useful only if R is small and $Vals(S.A)$ is large
(e.g., A is a primary key for S)

Zig-zag (two index based) natural join

Natural join $R \bowtie S$ using two indexes

Join the indexes on the join attributes A for both R and S:

1. identify relevant blocks of R and S using the indexes
2. load the blocks with the same join values
   and construct the output

Cost: at most $Blocks(R) + Blocks(S)$ (as one pass)

Requirement: blocks for each join value (for R and S together)
should be at most $MMBuffers$
(an estimation formula can be written
via $Vals(R.A)$ and $Vals(S.A)$
relying on the uniformity assumption)

Natural join algorithms: Comparison example

Natural join $R \bowtie S$ on a common attribute A,
both with clustered indexes

Statistics: $Tups(R) = 10,000$, $Blocks(R) = 1,250$, $Vals(R.A) = 100$
$Tups(S) = 5,000$, $Blocks(S) = 625$, $Vals(S.A) = 10$

	Cost (often estimation)	Requirement for *MMBuffers*
One-pass	1875	626
Nested-loop	9375	2
Two-pass sort-based	9375	36
Two-pass hash-based	5625	25
Hybrid hash-based	5019	25
One index-based on S	625,000	2
One index-based on R	62,500	2
Zig-zag	1875	76

Note: cost of nested-loop and hybrid hash-join depends on *MMBuffers*;
we take $MMBuffers = 101$

4. Summary

Algorithms summary

One-pass: good, but applies generally when one argument fits into the memory

Two-pass: much lighter memory requirements

- Sort-based:
  give sorted results, which can be utilised later
- Hash-based:
  usually requires less memory (only the smaller counts)
  relies on good hash function (equal distribution over buckets)
- Index-based:
  requires indexes
  great for selection
  often efficient for join

N-pass: generalisation of two-pass for even bigger relations

N-pass algorithms

N-pass: generalisation of two-pass for even bigger relations

Example: $Blocks(R) = 1,000,000$

- TPMMS requires $\sqrt{Blocks(R)} \leq MMBuffers$ so $MMBuffers \geq 1,000$
- for bigger R, two passes are not enough

N-PMMS sorting:

- recursive TPMMS
- Cost: $(2N - 1) * Blocks(R)$
- Requirement: $\sqrt[N]{Blocks(R)} \leq MMBuffers$

Similar ideas applies to hash-based algorithms

Back to query plans

Logical query plan: RA expression

Physical query plan: Logical query plan where an algorithm is assigned to each operation

For simplicity, we assumed that we have a logical plan constructed, and studies algorithms for RA operations 'in isolation' (often several algorithms for one operation with different costs and requirements)

In reality, these costs are used to choose logical plans
(even more search space)

We may have problems with evaluation:
our algorithms rely on assumptions, which may not hold in reality,
and we may have to use another algorithm
(or even adapt the already running one loosing the performance)

5. Query plans in real DBMSs

Overview

Real DBMSs estimate costs based on their knowledge on the system
(block size, main-memory operation costs, etc.)

For example, in PostgreSQL the defaults are something like

```
seq_page_cost = 1
random_page_cost = 4
cpu_tuple_cost = 0.01
cpu_operator_cost = 0.0025
cpu_index_tuple_cost = 0.005
```

Usually updated automatically,
but sometimes can be manually changed (for estimation)

PostgreSQL query plans analysis

PostgreSQL has `EXPLAIN` command that provides information on query plans and their costs (Play with it, see accessible manual at `https://wiki.postgresql.org/wiki/Using_EXPLAIN`)

Example: `EXPLAIN SELECT * FROM Post ORDER BY Body LIMIT 50;`

```
Limit (cost = 23283.24..23283.37 rows = 50 width = 422)
-> Sort (cost = 23283.24..23859.27 rows = 230412 width = 422)
Sort Key:  body
-> Seq Scan on post (cost = 0.00..15629.12 rows = 230412 width = 422)
```

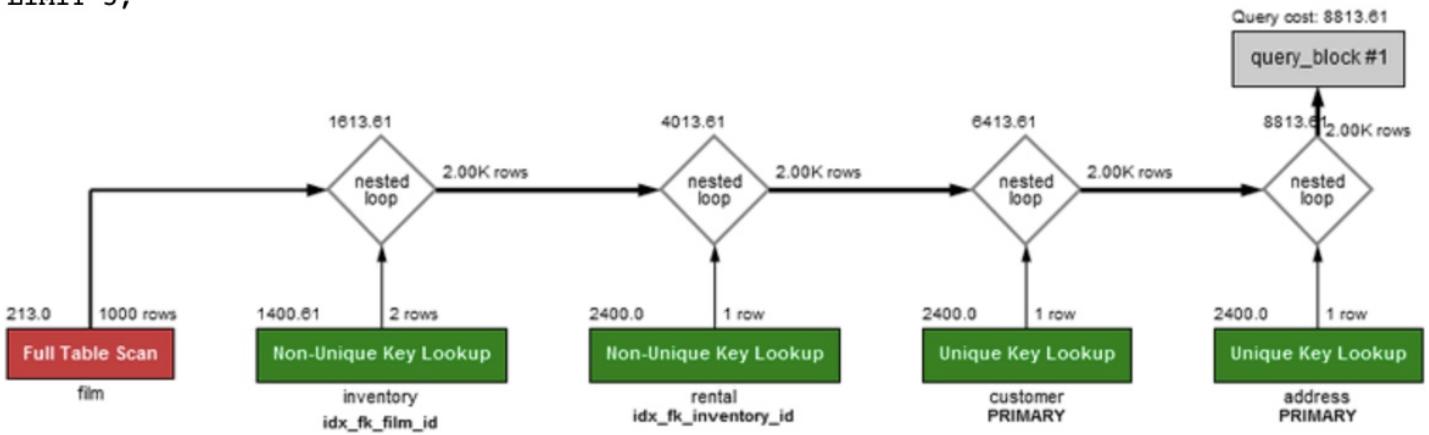
Has parameters:

- `VERBOSE`: more information
- `ANALIZE`: also runs the query and compares plan with reality
- ...

Quickly become unreadable, but there are visualisation tools (e.g., `http://tatiyants.com/pev/`)

MySQL query plan analysis

```
SELECT CONCAT(customer.last_name, ', ', customer.first_name) AS customer, address.phone, film.title
FROM rental INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
WHERE rental.return_date IS NULL
AND rental_date + INTERVAL film.rental_duration DAY < CURRENT_DATE()
LIMIT 5;
```



Much less information

What have we learned in Part 1?

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

$\downarrow$ parsing

(parse tree)

$\downarrow$ compilation

$$\pi_{title}(\sigma_{starName=name}(\mathbf{StarIn} \times \pi_{name}(\sigma_{birthDate\ LIKE\ '\%1960'}(\mathbf{MovieStar}))))$$

$\downarrow$ optimisation

$$\pi_{title}(\mathbf{StarIn} \bowtie_{starName=name} \pi_{name}(\sigma_{birthDate\ LIKE\ '\%1960'}(\mathbf{MovieStar})))$$

$\downarrow$ evaluation over tables

(Result)