DBM1
Part 3: Database internals

Vincent Primault

vincent.primault@insa-lyon.fr

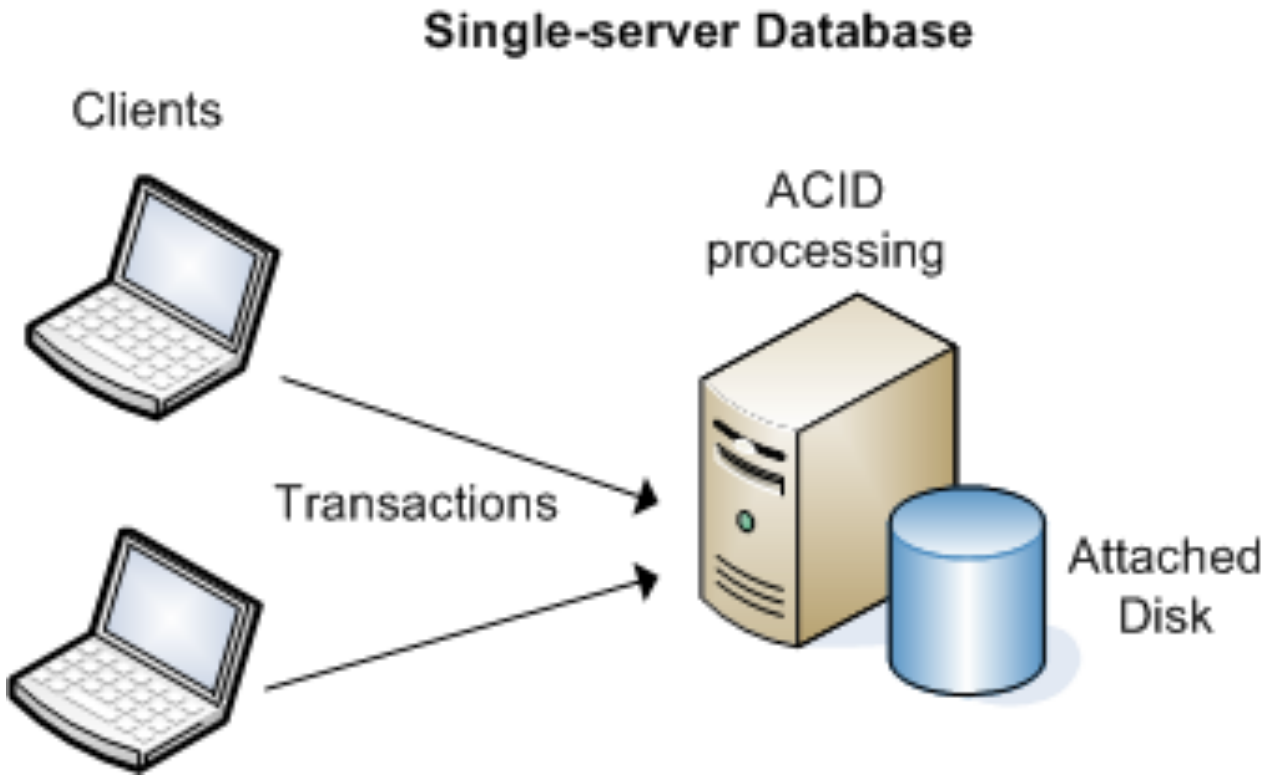
Course outline

- ~~Databases fundamentals~~ **Done!**
- ~~Relational algebra~~ **Done!**
- ~~SQL language~~ **Done!**
- Database internals **Today**
- Distributed databases & NoSQL

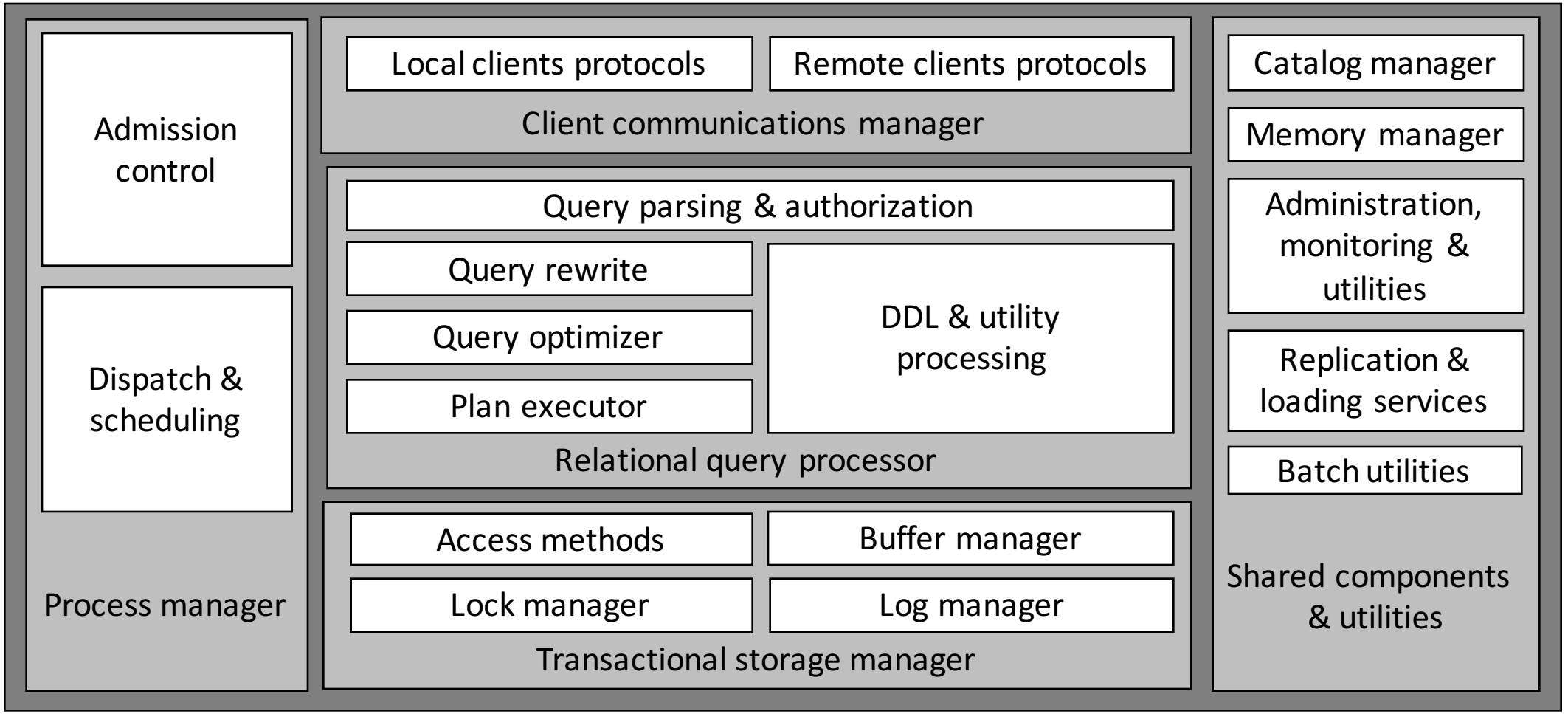
Parts of SQL we will not see

- Data modification statements (**INSERT**, **UPDATE**, **DELETE**…).
- Data description language (**CREATE TABLE**...).
- Complex data types such as date types or geographical types.
- Other structures such as views or materialized views.
- More about functions.

Client/server architecture



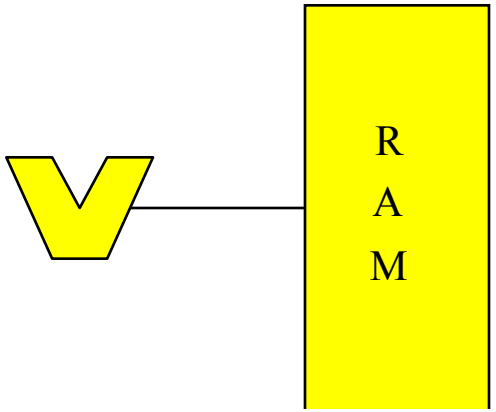
DBMS architecture



I/O predominance

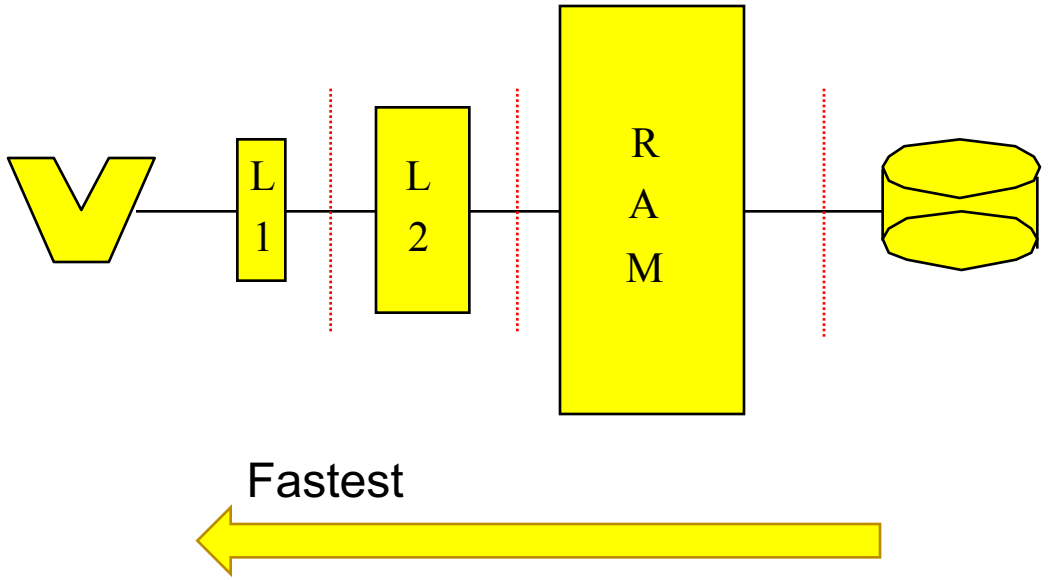
DBM1 – Part 3: Database internals

Random Access Machine Model



- Standard theoretical model of computation:
  - Infinite memory
  - Uniform access cost
- **Simple model crucial for success of computer industry.**

Hierarchical Memory



- Modern machines have complicated memory hierarchy:
  - Levels get **larger** and **slower** further away from CPU
  - Data moved between levels using **large blocks**

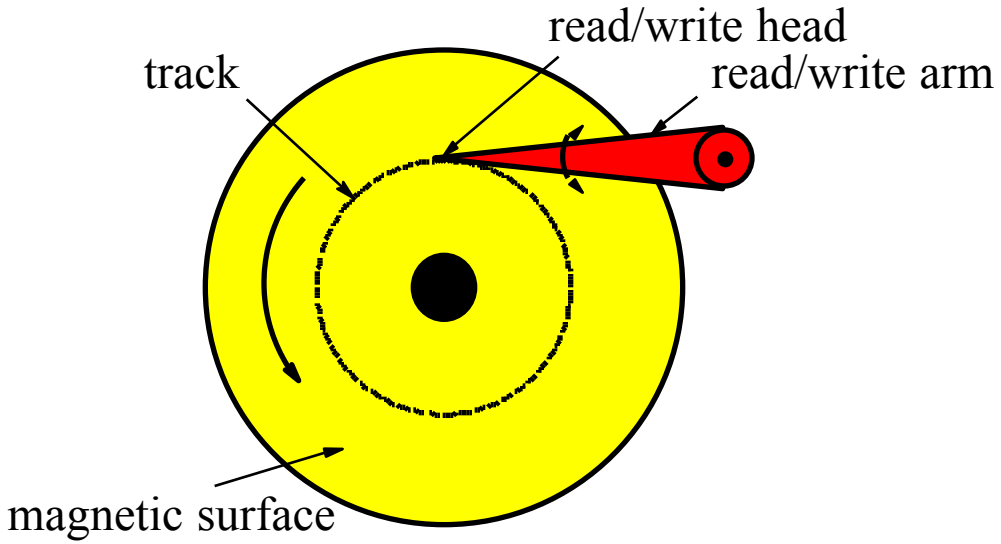
Slow I/O

Disk access is 10^6 times slower than main memory access.

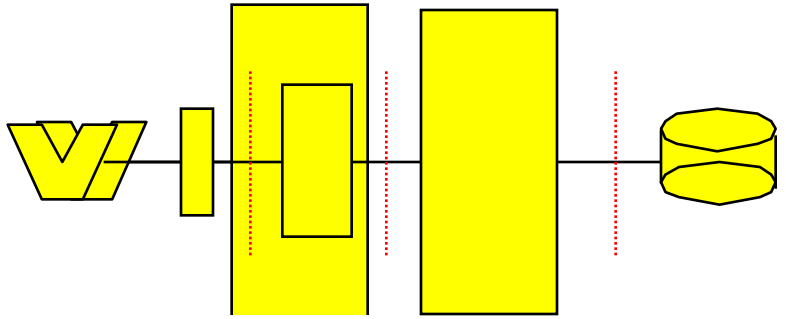
"The difference in speed between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk." (D. Comer)

Slow I/O (cont'd)

- Disk systems try to amortize large access time transferring large contiguous blocks of data (8-16 Kb).
- Important to store/access data to take advantage of blocks (locality).



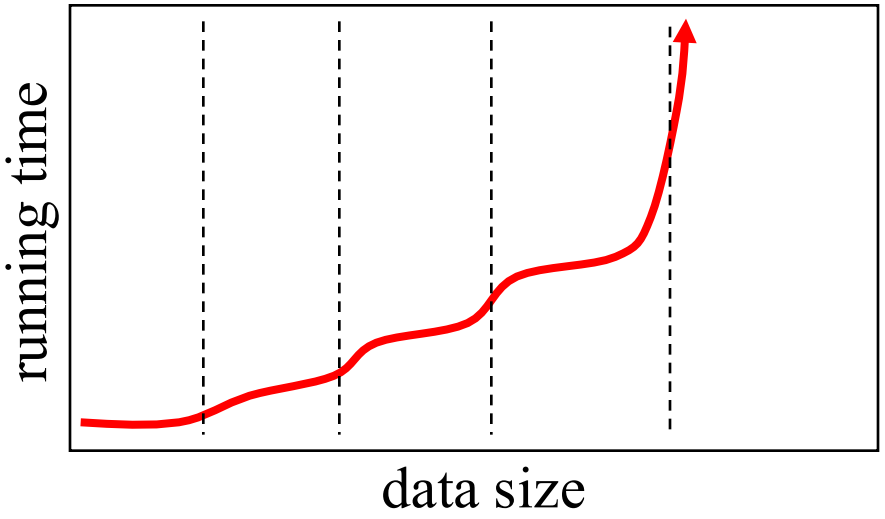
Scalability problems



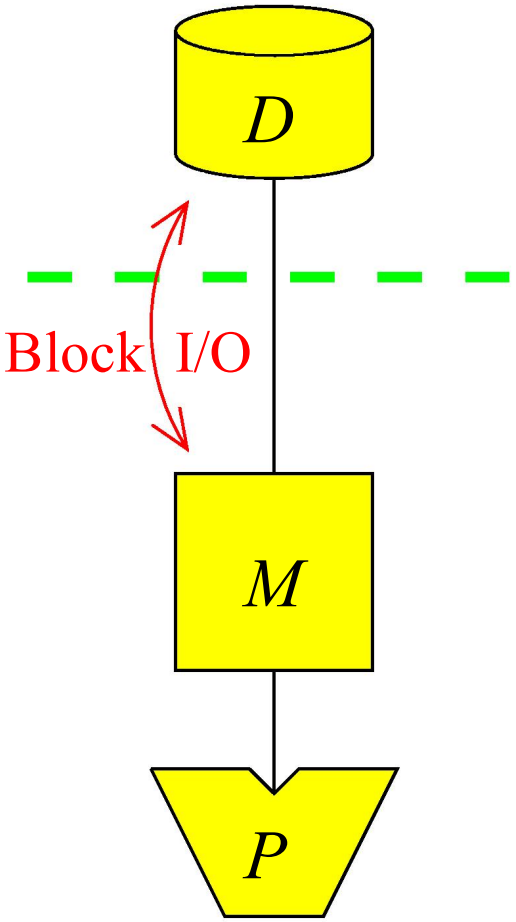
- Most programs are developed with in RAM-model
- They run on large datasets because OS moves blocks as needed
- Moderns OS utilizes sophisticated paging and prefetching strategies
  - But if program makes scattered accesses even good OS cannot take advantage of block access

⇓

Scalability problems!



External memory model



N = # of items in the problem instance

B = # of items per disk block

M = # of items that fit in main memory

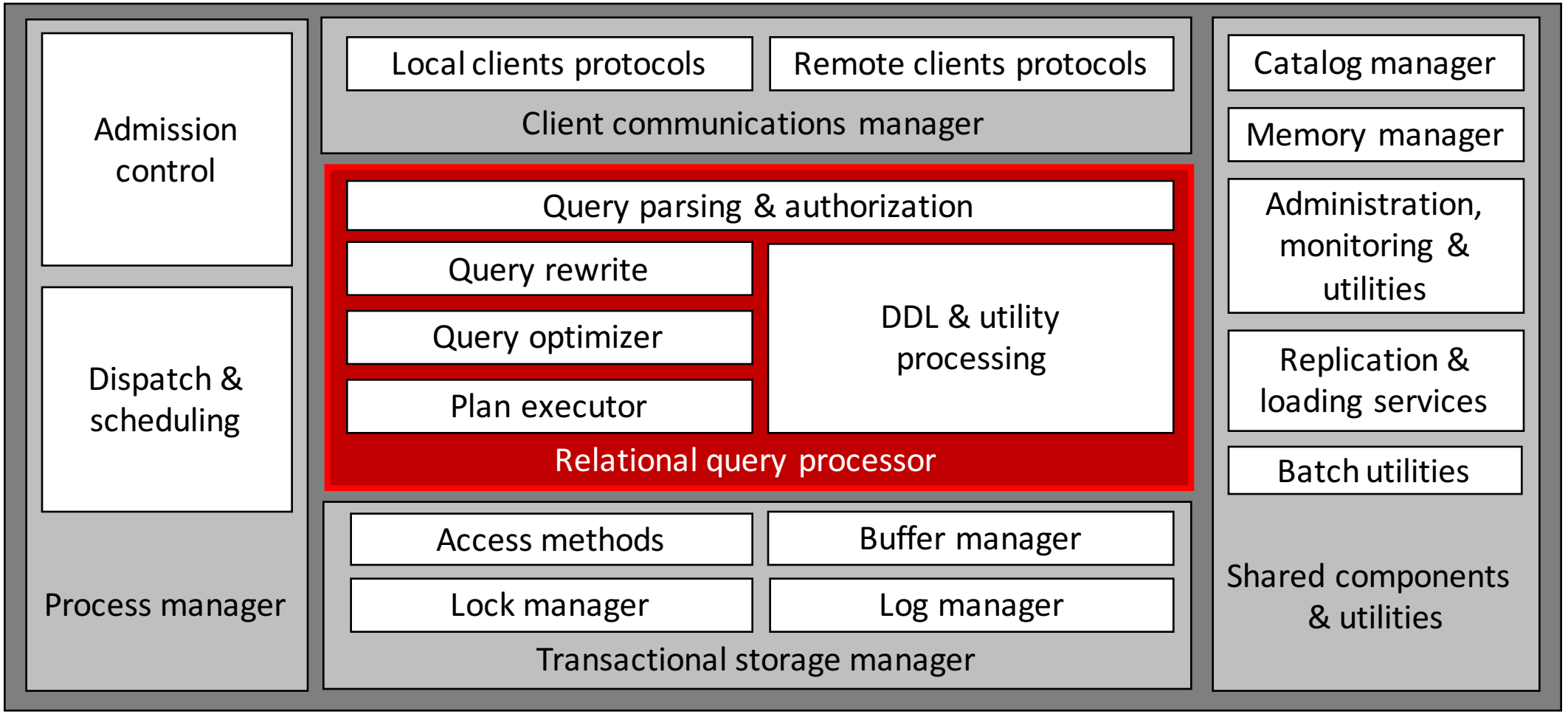
I/O: Move block between memory and disk

=> Out-of-core algorithms: taking into account "*memory swap*" to design them

Query processing

DBM1 – Part 3: Database internals

DBMS architecture



Basic notions through an example

Give the values of B and D such that A=c, E=2 and R.C=S.C.

Algebra: $\pi_{B,D}(\sigma_{A=c \text{ AND } E=2}(R \bowtie S))$

SQL:

```
SELECT B,D
FROM R
JOIN S ON R.C = S.C
WHERE R.A = 'c' AND S.E = 2
```

R

A	B	C
a	1	10
b	1	20
…	…	…

S

C	D	E
10	x	2
20	y	2
…	…	…

Executing the query

Algebra: $\pi_{B,D}(\sigma_{A=c \text{ AND } E=2}(R \bowtie S))$

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

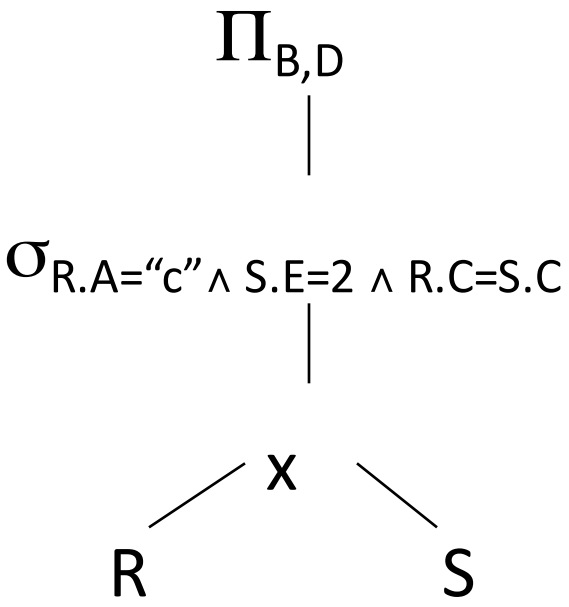
S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

B	D
2	x

Answer

Plan #1

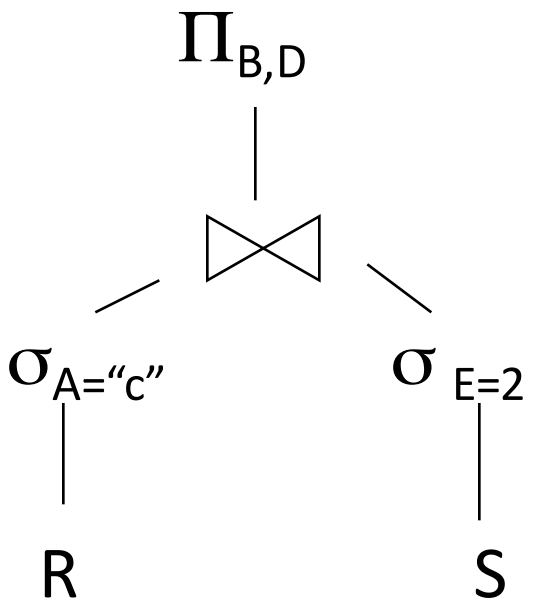


Plan #1 (cont'd)

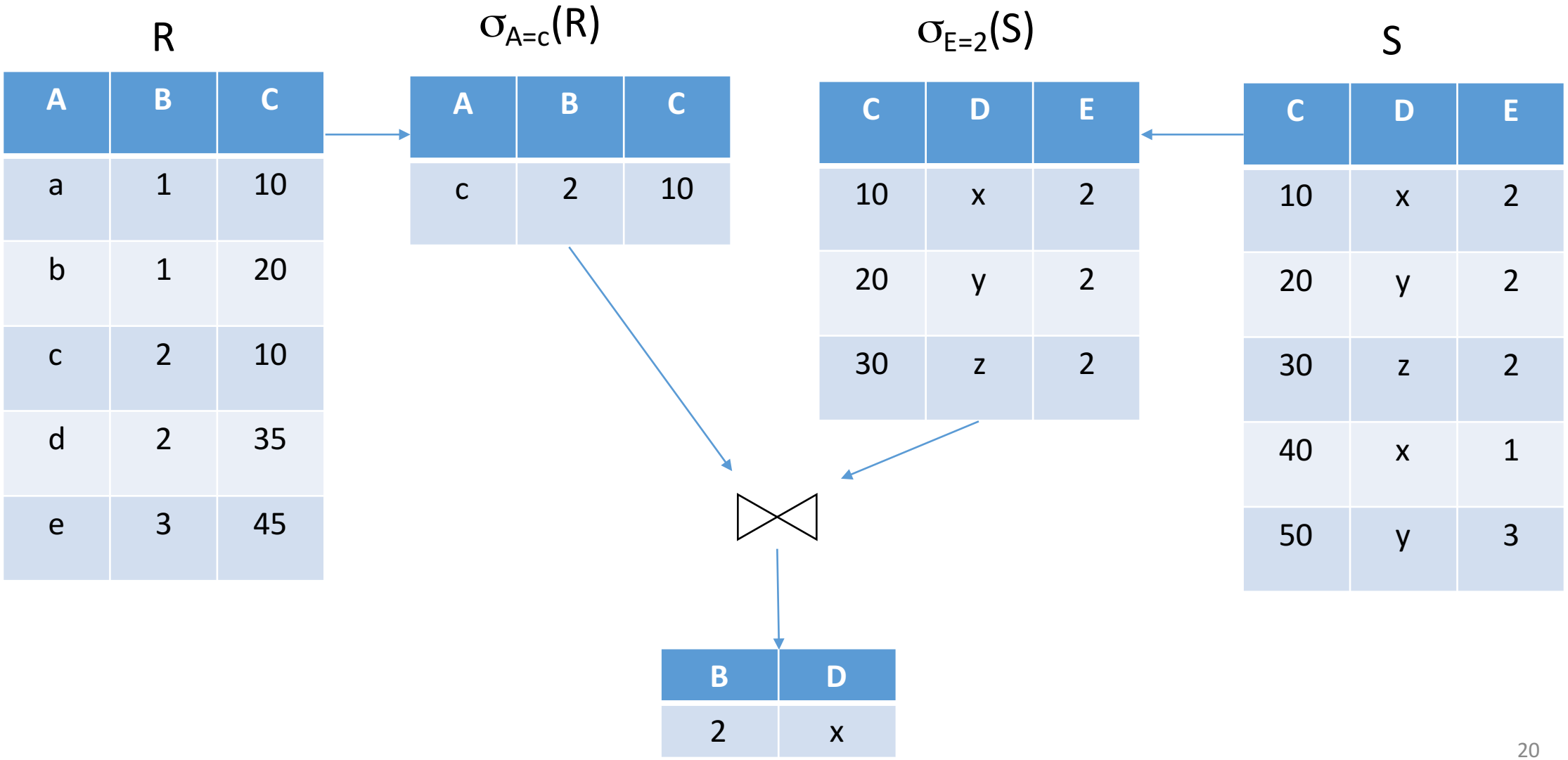
R.A	R.B	R.C	S.C	S.D	S.E
a	1	10	10	x	2
a	1	10	20	x	2
…	…	…	…	…	…
c	2	10	10	x	2
…	…	…	…	…	…

This enumerates every tuple from the cross product, just to find one row at the end...

Plan #2



Plan #2 (cont'd)


R
A B C
a 1 10
b 1 20
c 2 10
d 2 35
e 3 45
σA=c(R)
A B C
c 2 10
σE=2(S)
C D E
10 x 2
20 y 2
30 z 2
S
C D E
10 x 2
20 y 2
30 z 2
40 x 1
50 y 3
⋈
B D
2 x

Plan #3

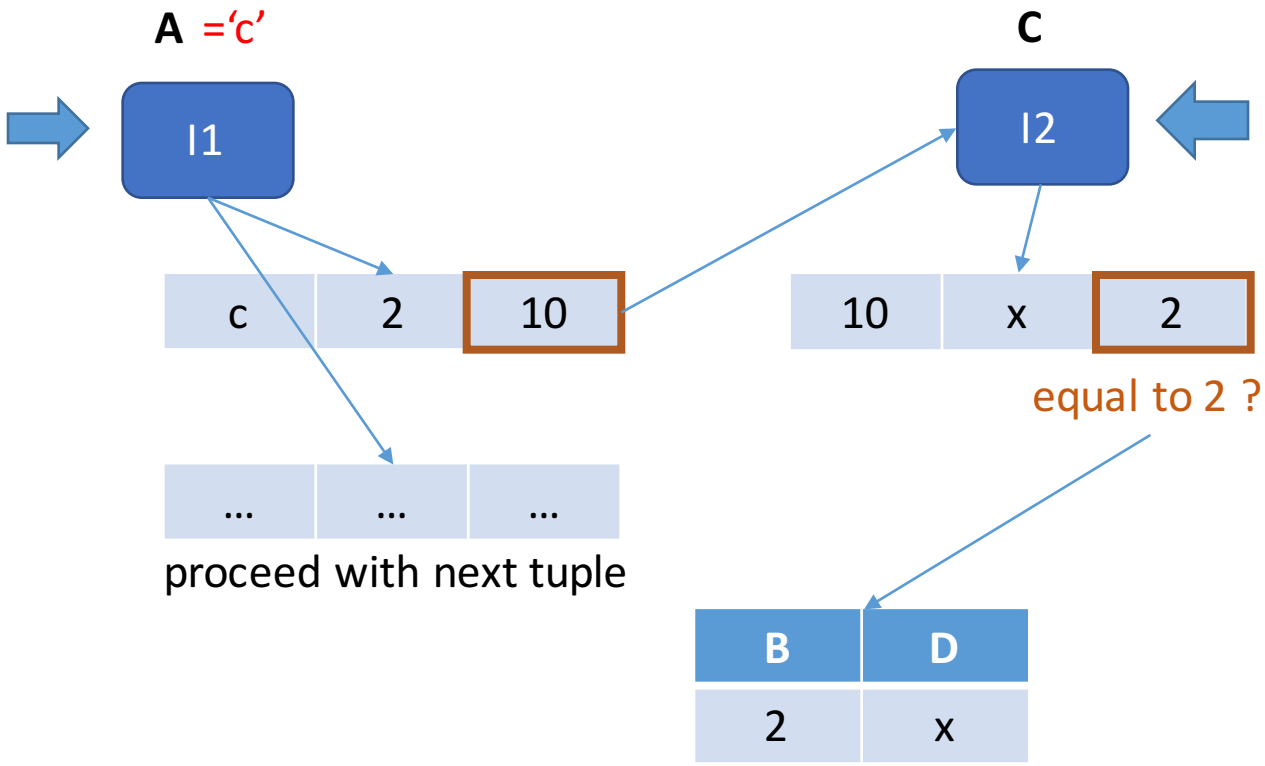
Given that we have indexes on R.A and S.C.

1. Use R.A index to select R tuples with R.A = “c”
2. For each R.C value found
    1. Use S.C index to find matching tuples
    2. Eliminate S tuples S.E ≠ 2
    3. Join matching R,S tuples
    4. Project on B,D attributes

Plan #3 (cont'd)

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

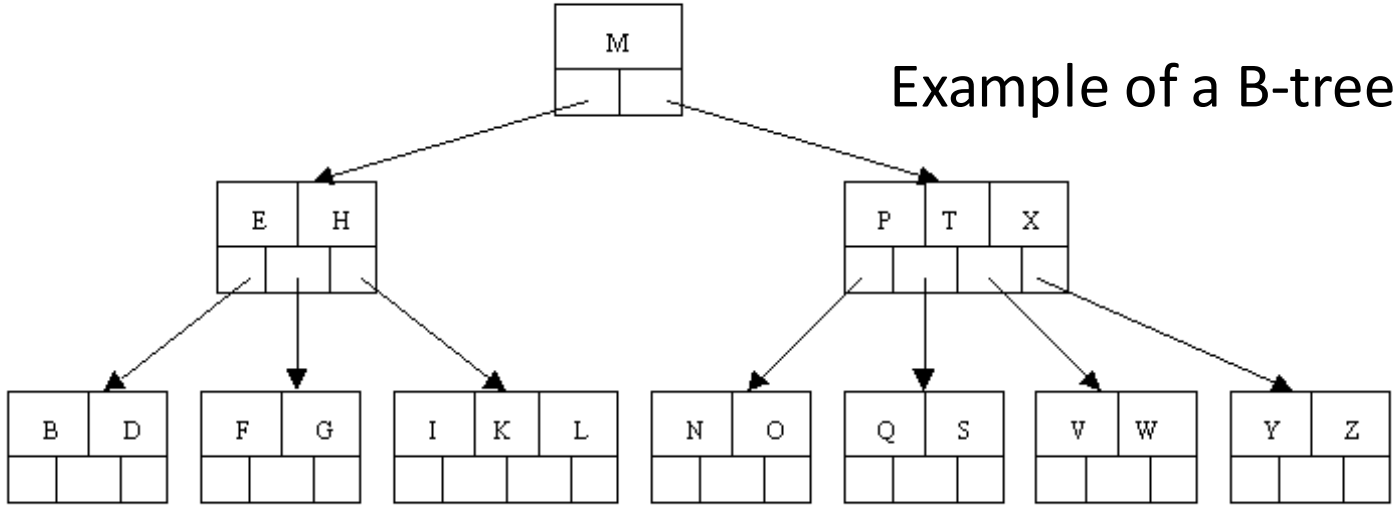


S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

A quick word about indexes

- Without any index, looking for tuples of R where A=c is an O(n) operation: you have to read each tuple and test the condition.
- With an index, you can have an O(log(N)) performance, and possibly achieve an O(1) under some assumptions.



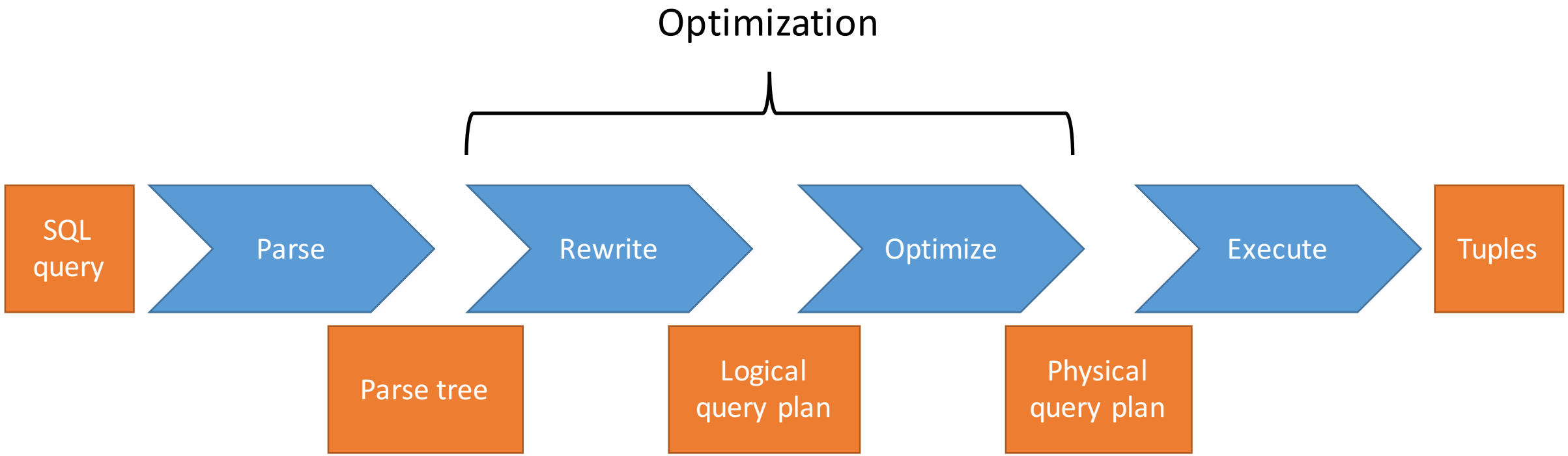
Example of a B-tree

Synthesis

- Plans #1 and #2 come from the relational algebra: the same query can be rewritten again and again.
- Plan #3 is a refinement of plan #2: plan #2 + indexes + some algorithmic tricks.

=> An example of a *physical query plan* versus a *logical query plan*.

Query processing



1. Parse phase

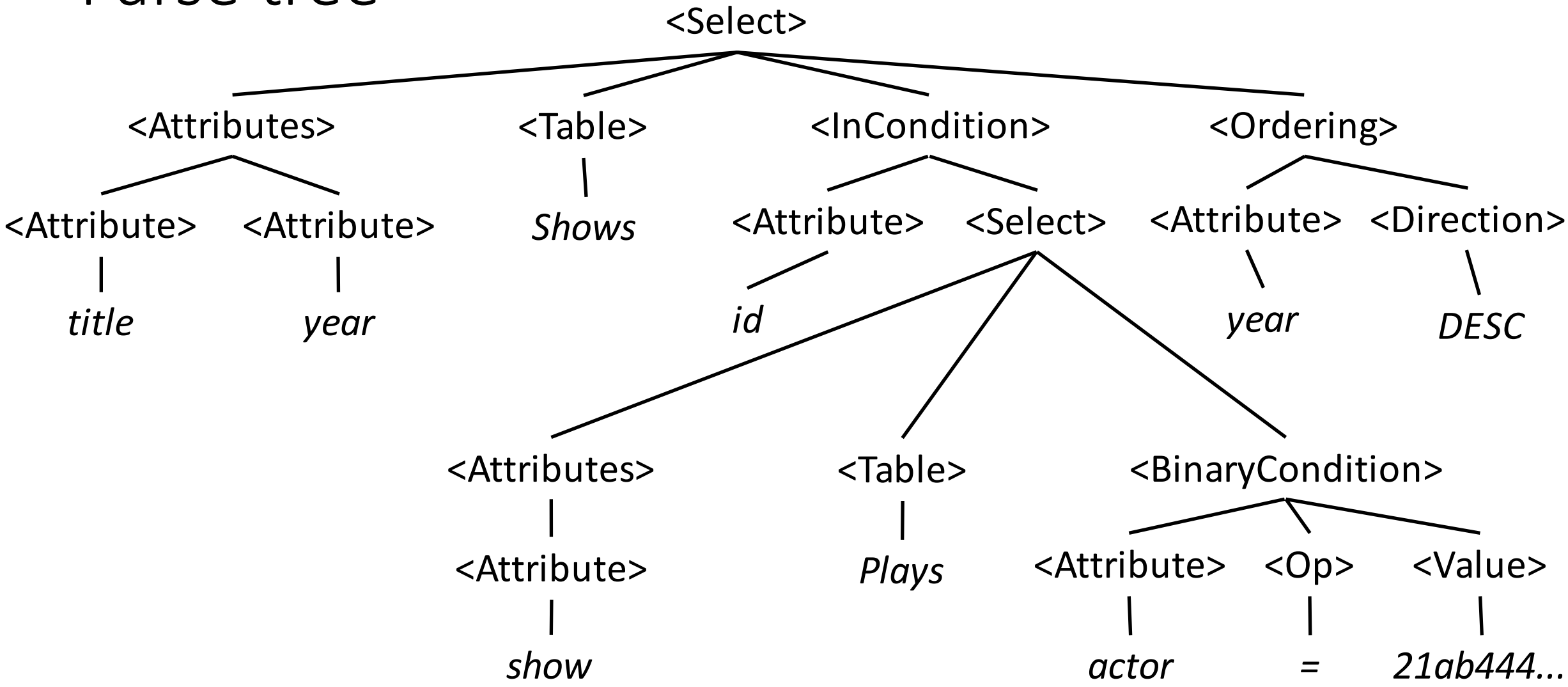
- Parses the query from a plain text string into a parse tree.
- The parser checks the query is correct.
  - *Syntax:* the grammar is correct.
  - *Semantic:* the identifiers mention valid objects. Operations must be consistent w.r.t. data types.
  - *Authorization:* the user must have access to the mentioned objects.

An example SQL query

```
SELECT title, year
FROM Shows
WHERE id IN (
SELECT show
  FROM Plays
  WHERE actor = '21ab444ee2d376dd45dc'
)
ORDER BY year DESC;
```

What does this query do?

Parse tree


<Select>
<Attributes>
<Table>
<InCondition>
<Ordering>
<Attribute>
<Attribute>
Shows
<Attribute>
<Select>
<Attribute>
<Direction>
title
year
id
year
DESC
<Attributes>
<Table>
<BinaryCondition>
<Attribute>
Plays
<Attribute>
<Op>
<Value>
show
actor
=
21ab444...

2. Rewrite phase

- Translates the parse tree into a logical query plan.
- The logical query plan is often represented a relational algebra expression => need to use extended relational algebra to represent each SQL operator!
- Rewrites the relational algebra to transform it.

Rewriting relational algebra

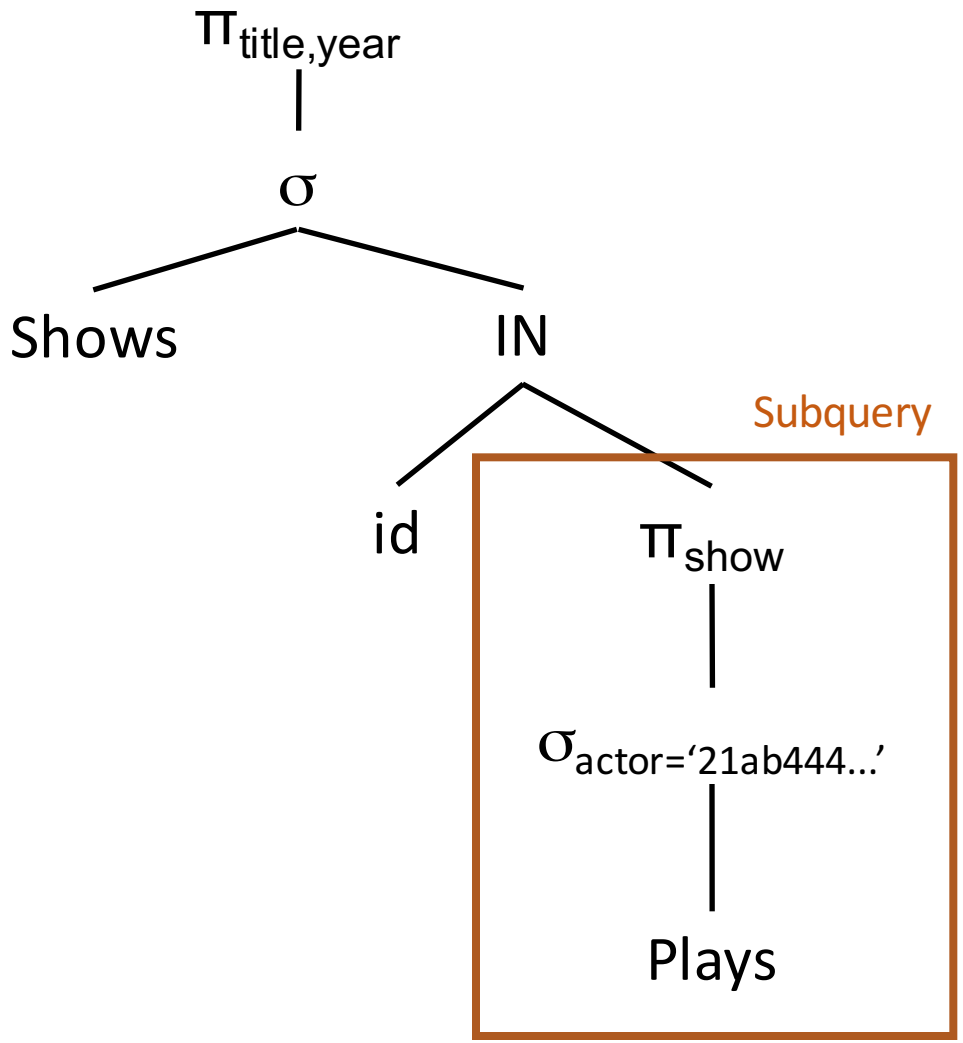
- *Views expansion.* Recursively replaces views with their definition, in order to express the query only with tables and attributes.
- *Constant arithmetic evaluation.* Simplifies constant arithmetic expressions, e.g., 10+2 = 12.
- *Logical rewriting of predicates.* Simplifies predicates in the **WHERE** clause, e.g., NOT x > 10 ⇔ x <= 10, x < 10 AND x > 100 ⇔ *False*. An unsatisfiable predicate can cause a query to immediately return an empty result.
- *Semantic optimization.* Takes into account the integrity constraints stored in the catalog.
- *Heuristic rewrites.* Based on the relational algebra to preserve equivalence of logical query plans.

Heuristic rewrite

- Most of the time, optimizers will operate on individual SELECT-FROM-WHERE blocks, and not across blocks.
- Flattening nested queries when possible reduces the number of these blocks, ultimately to a single block.
- Early selection and projection are usually good (i.e., pushing them down in the tree).
- AND selections can be splitted and handled separately.
- New projections can be adedd.
- No transformation is always good !

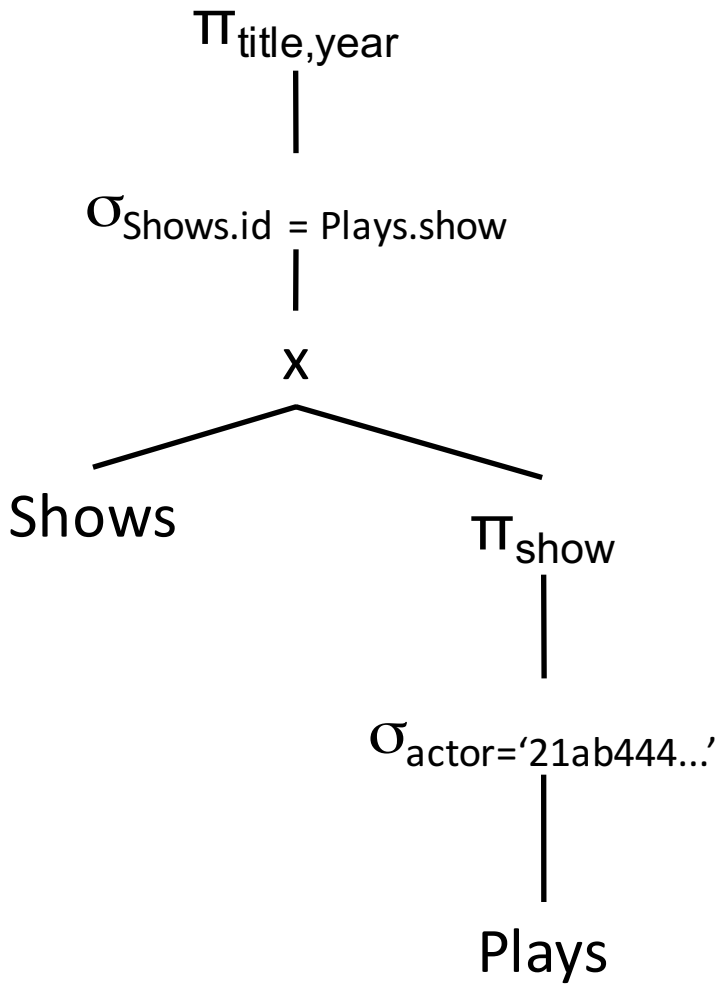
Generating relational algebra

Something between a parse tree and a relational algebra query

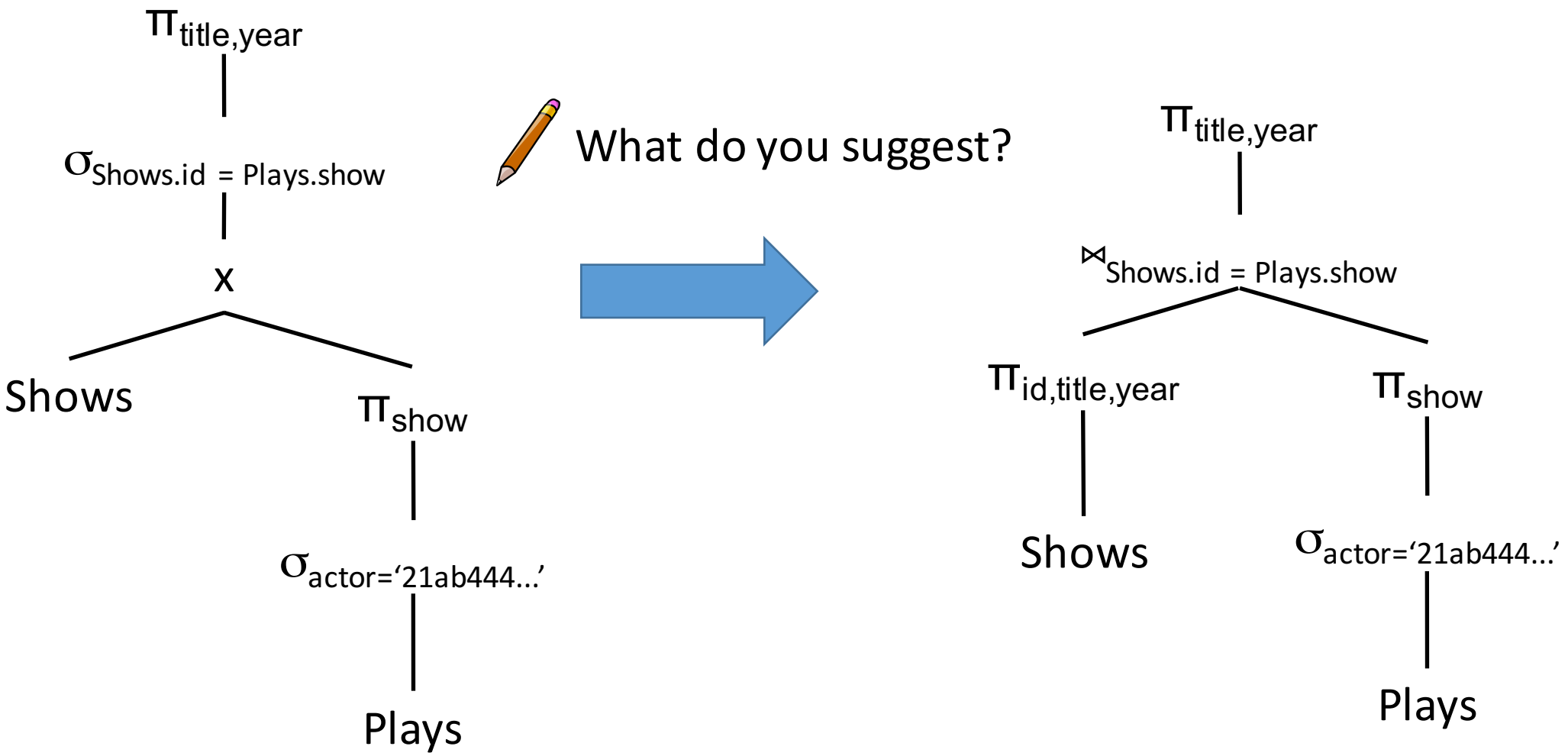


Logical query plan

Standard transformation for IN conditions in SQL



Improved logical query plan



From a *logical* to a *physical* query plan

- For each operator in the logical plan, pick one available implementation
  - For the join, can be nested loops join, hash join, sort-merge join…
- Define additional implementations for implicit actions in the logical plan
  - Scanning, sorting…
- Define the explicit behavior between two operators.
  - Store the intermediate results on disk.
  - Use iterators and send an argument on main-memory buffer at a time (pipeline).

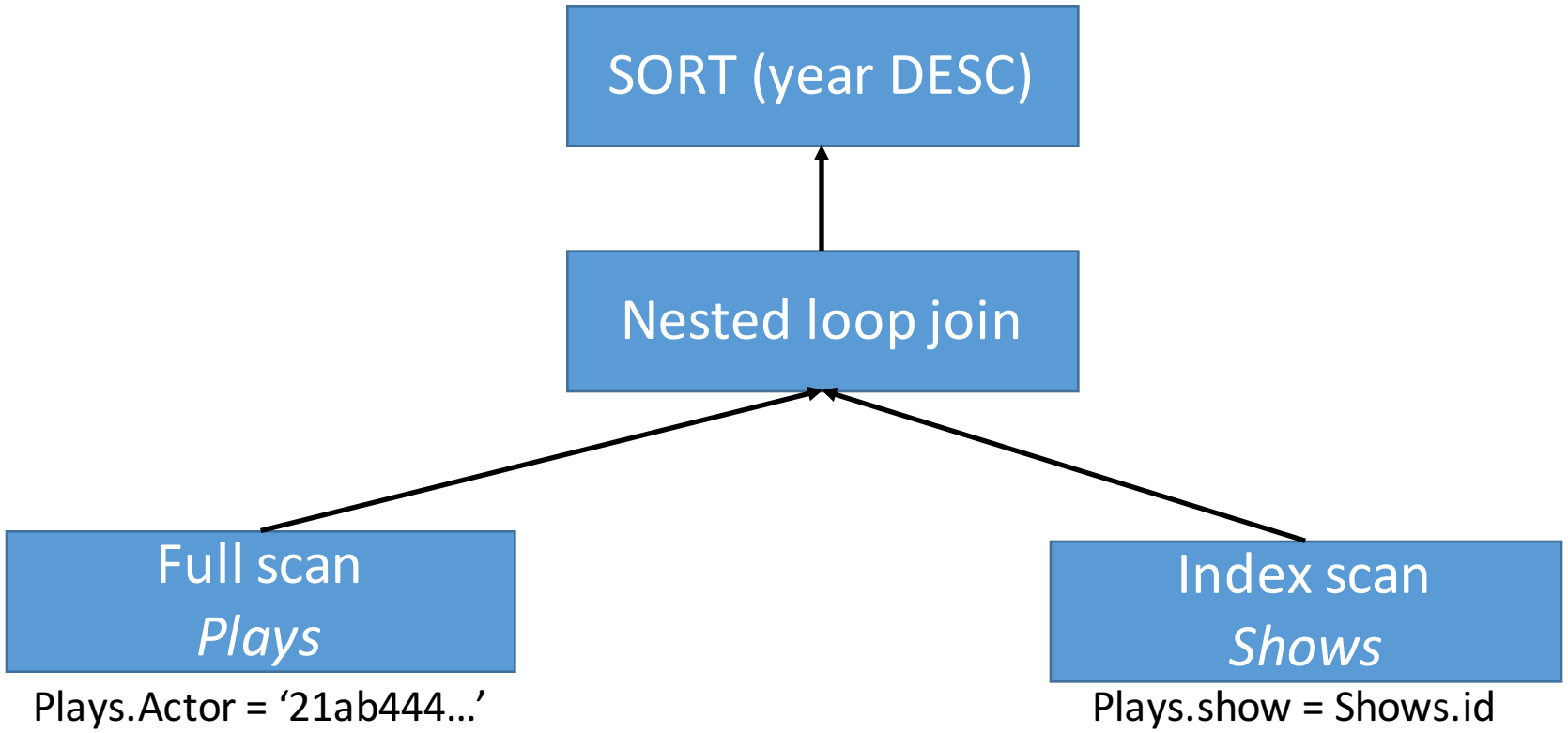
3. Optimization phase

- The query optimizer's goal is to transform a *logical* query plan into a *physical* query plan.
- Many physical plans will match a given logical plan. The optimizer tries to pick the best one, by using a cost model based on data statistics.
- The query plan can be represented in several ways, e.g., compiled machine code, interpretable plan, list of op-codes.

Consequences of I/O predominance

- In the scope of DBMS, given a SQL query, query optimization has to find a compromise between
  - CPU (classical complexity of algorithms)
  - I/O (complexity for out-of-core algorithms)
- In different contexts, compromise may imply other dimensions, for instance
  - QoS
  - Network latency
  - Cost of service invocations…

Physical query plan



On the number of possible plans

- Can be huge!
- Exponential number of plans due to:
  - Rewriting rules.
  - Existing implementations of physical operators.

- So, we have to define two components:

1. a cost model to compare plans.
2. a strategy to get an “optimal” physical plan from the plan space.

Estimating cost of query plan

1. Estimating size of results.
2. Estimating the number of I/Os.

- Estimating result size: Keep statistics for relation R
  - T(R) : # tuples in R
  - S(R) : # of bytes in each R tuple
  - B(R) : # of blocks to hold all R tuples
  - V(R, A) : # distinct values in R for attribute A
- Cost model: has to be computed quite efficiently!

Estimating cost of a query plan (cont'd)

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

A: 20 byte string

B: 4 byte integer

C: 8 byte date

D: 5 byte string

T(R) = 5

V(R,A) = 3

V(R,B) = 1

S(R) = 37

V(R,C) = 5

V(R,D) = 4

T(R): # tuples in R
S(R): # bytes in each tuple of R
B(R): # blocks to hold all tuples of R
V(R,A): # distinct values of A in R

Compute T(R), S(R), V(R,A), V(R,B), V(R,C), V(R,D).

Size estimates for W = R1 x R2

- T(W) = T(R1) x T(R2)
- S(W) = S(R1) + S(R2)

T(R): # tuples in R
S(R): # bytes in each tuple of R
B(R): # blocks to hold all tuples of R
V(R,A): # distinct values of A in R

Size estimate for $W = \sigma_{Z=val}(R)$

- S(W) = S(R)
- T(W) = ?

- Assumption: values in expression *Z=val* are uniformly distributed over possible V(R,Z) values.

T(R): # tuples in R
S(R): # bytes in each tuple of R
B(R): # blocks to hold all tuples of R
V(R,A): # distinct values of A in R

Size estimate for $W = \sigma_{Z=val}(R)$ (cont'd)

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

V(R,A)=3

V(R,B)=1

V(R,C)=5

V(R,D)=4

- $T(W) = T(R)/V(R, Z)$
- $T(\sigma_{A=val}(R)) = T(R)/V(R, A) = 5/3$
- $T(\sigma_{B=val}(R)) = T(R)/V(R, B) = 5/1 = 5$
- $T(\sigma_{C=val}(R)) = T(R)/V(R, C) = 5/5 = 1$
- $T(\sigma_{D=val}(R)) = T(R)/V(R, D) = 5/4$

T(R): # tuples in R
S(R): # bytes in each tuple of R
B(R): # blocks to hold all tuples of R
V(R,A): # distinct values of A in R

Size estimate for W = R1 ⋈ R2

Let X = attributes of R1, Y = attributes of R2.

- If X ∩ Y = ∅, then same as R1 x R2.
- If X ∩ Y = A, then

T(R): # tuples in R
S(R): # bytes in each tuple of R
B(R): # blocks to hold all tuples of R
V(R,A): # distinct values of A in R

Site estimate for T(W = R1 ⋈ R2) with X ∩ Y = {A}

- S(W) = S(R1) + S(R2) – S(A)
- Assumption:
  - V(R1,A) ≤ V(R2,A) ⇒ Every A value in R1 is in R2
  - V(R2,A) ≤ V(R1,A) ⇒ Every A value in R2 is in R1

- When V(R1,A) ≤ V(R2,A), each tuple of R1 matches with T(R2)/V(R2, A) tuples.
- So $T(W) = \frac{T(R2) \,.\, T(R1)}{V(R2, A)}$

T(R): # tuples in R
S(R): # bytes in each tuple of R
B(R): # blocks to hold all tuples of R
V(R,A): # distinct values of A in R

Size estimate for T(W = R1 ⋈ R2) with X ∩ Y = {A}

- $\mathrm{T(W)} = \frac{T(R2)\ .T(R1)}{\max(V(R1,A),V(R2,A))}$

- These principles can be extended to **other operators**.

T(R): # tuples in R
S(R): # bytes in each tuple of R
B(R): # blocks to hold all tuples of R
V(R,A): # distinct values of A in R

Statistics-based cost model

- We saw a statistical model, based upon uniformity and independance assumptions. They are usually not valid…
- …but allow simple (thus efficient) computations.
- The database has to maintain stats about T(R), S(R), B(R) and V(R, A) for each table.
- More accuracy can be provided with histograms to approximate more closely value distributions.

Sampling-based cost model

- Gather necessary characteristics of a query plan (input relations and intermediate results) by running the query on a small sample and by extrapolating to the full input size at query execution time.

- What the size of the sample should be? Trade-off between
  - Efficiency: it should take far less time than the query to execute.
  - Accuracy: it should be large anough to have a good prediction.

Search strategy in plan space

- How to enumerate and select an optimal plan w.r.t. a given cost model?
- Beautiful optimization problem!
  - The computation of an "optimal" plan has to be performed efficiently
- Almost all existing optimization techniques could be applied:
  - Heuristics
  - Randomized search algorithms
  - Top-down (like Vulcano, Cascades (in SQL server)) or Bottom-up
    - Dynamic programming (pruning of useless sub-trees)
    - Selinger-style (or System-R, ancestor of IBM DB2)

4. Execution phase

- Given a fully specified execution plan, it is now possible to execute it and get our tuples.
- The role of the executor depends on the way the execution plan is represented.
  - If it is compiled to machine code, the executor is basically an interpreter.
  - If it is an interpretable representation, the executor calls procedures for each operator in the plan.
  - We focus on the latter case.

Iterators

- All operators in a query plan are implemented as iterators.
- Each operator is thus independant from its parents or children in the graph.
- Because of the *next()* method, execution must be done in a single thread.

```
class Iterator
{
    List<Iterator> inputs;
    void init();
    Tuple getNext();
    void close();
}
```

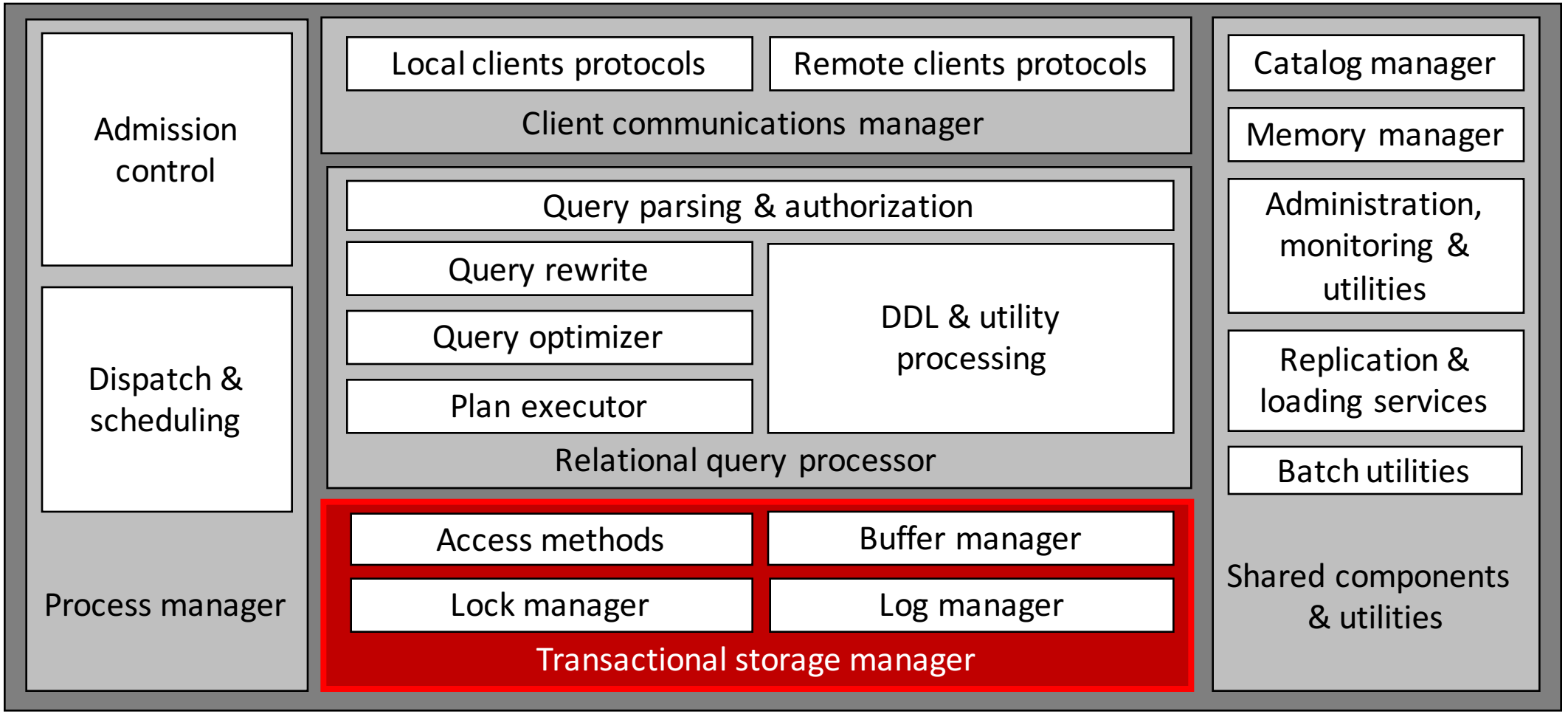
Synthesis

- Relational algebra is useful!
- Optimizers are a very complex piece of software.
- They are able to optimize badly written queries... but only to a certain extent. The basis of their work remains what you wrote.
- Never forget: "No one knows what the best execution plan for a given query is".

Transactional model

DBM1 – Part 3: Database internals

DBMS architecture



Transactions

- It is a unit of work that has to be treated in a coherent and reliable way, inpendently from other transactions.
- This unit of work must be either entirely executed or not executed, there cannot be an intermediate valid state.
- It is mainly useful for queries modifying the data.

- Typical example: moving money between bank accounts.

Transaction processing

1. Begin the transaction.
2. Execute a set of data manipulations and/or queries.
3. If no error occurs then commit the transaction.
4. If errors occur then rollback (= cancel) the transaction.

- In SQL implemented with **BEGIN**, **COMMIT** and **ROLLBACK** operators.

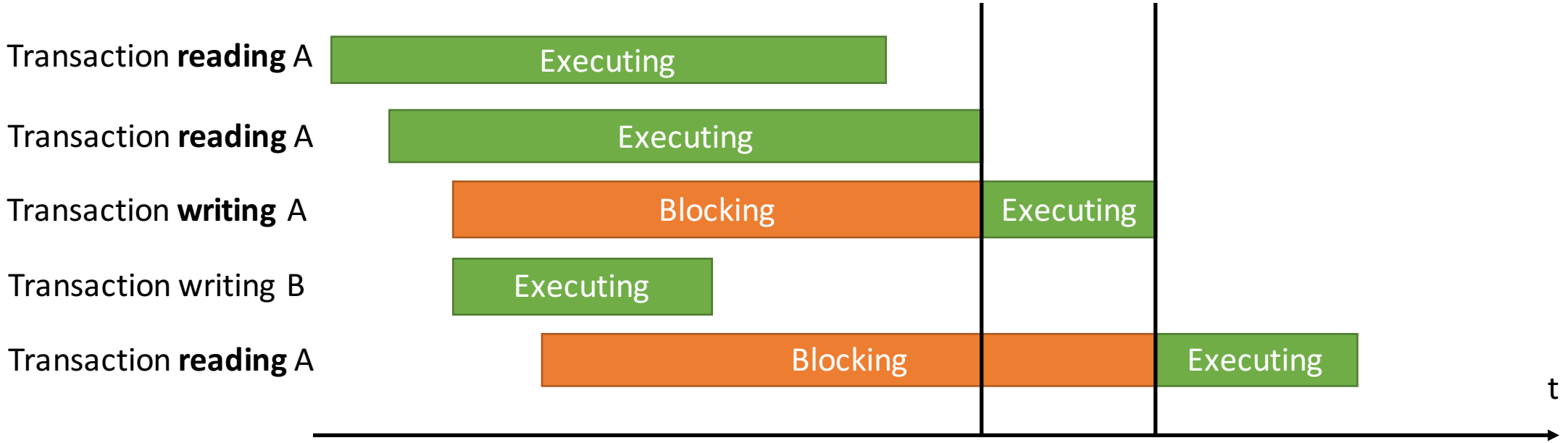
ACID properties

- **Atomic:** all-or-nothing operation.
- **Consistent:** constraints are respected.
- **Isolated:** processed independently from other transactions.
- **Durable:** committed transactions will survive permanently.

- Durability is guaranteed by locking and recovery.
- Atomicity and isolation are guaranteed by locking and logging.
- Consistency is managed by runtime checks on the query executor.

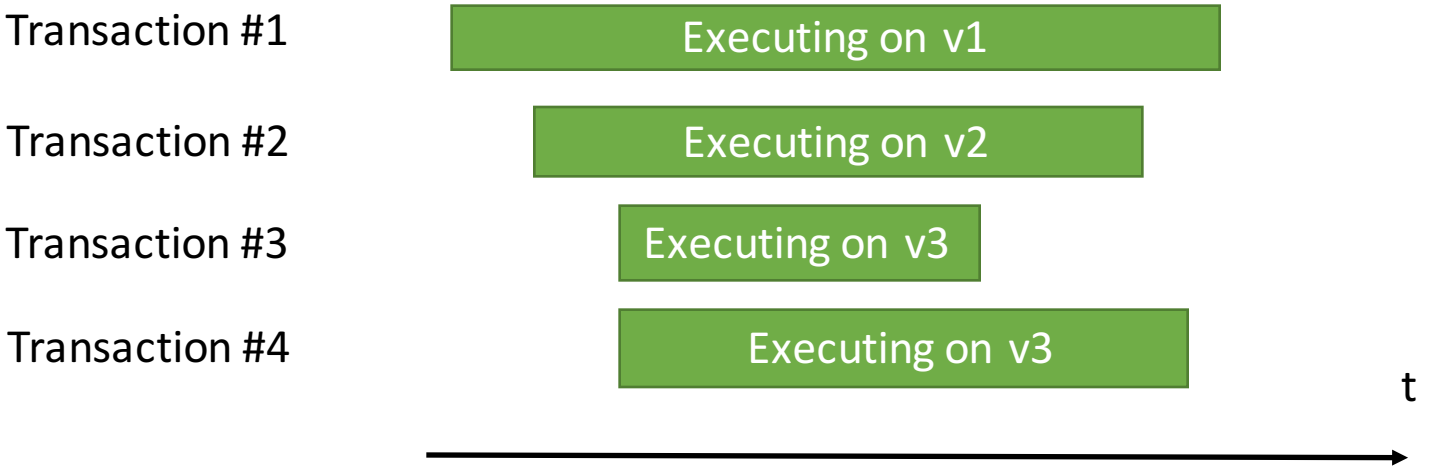
Strict two-phase locking

- Shared lock to read, exclusive lock to write.
- Locks are held until the end of the transaction, and acquiring a lock blocks until it is available.



Multi-version concurrency control

- No lock, but transactions are guaranteed to have a consistent view of the database in the past.



Optimistic concurrency control

- Transactions do not block, but maintain history of reads and writes. If there is a conflict when committing, one of them is rolled back.

Locking

- A lock is simply a name following naming conventions and representing either physical items (e.g., disk pages) or logical items (e.g., tuples).
- Locks come in different lock modes.
- The lock manager maintains a list of lock names, information about locks, and a waiting queue of other transactions.
- The lock manager is continuously detecting deadlocks.

Logging

Write-Ahead Logging protocol:

1. Each modification to a database page should generate a log record, and the log record must be flushed to the log device before the database page is flushed
2. Log records must be flushed in order; log record *r* cannot be flushed until all log records preceding *r* are flushed.
3. Upon a transaction commit request, a commit log record must be flushed to the log device **before** the commit request returns successfully.

Transactions can be undone ⇒ Atomicity

Transactions can be redone ⇒ Durability

Synthesis

- Transactions are a central component of a modern DBMS.
- They should guarantee atomicity, consistency, isolation and durability.
- These properties are guaranteed via locking and logging.