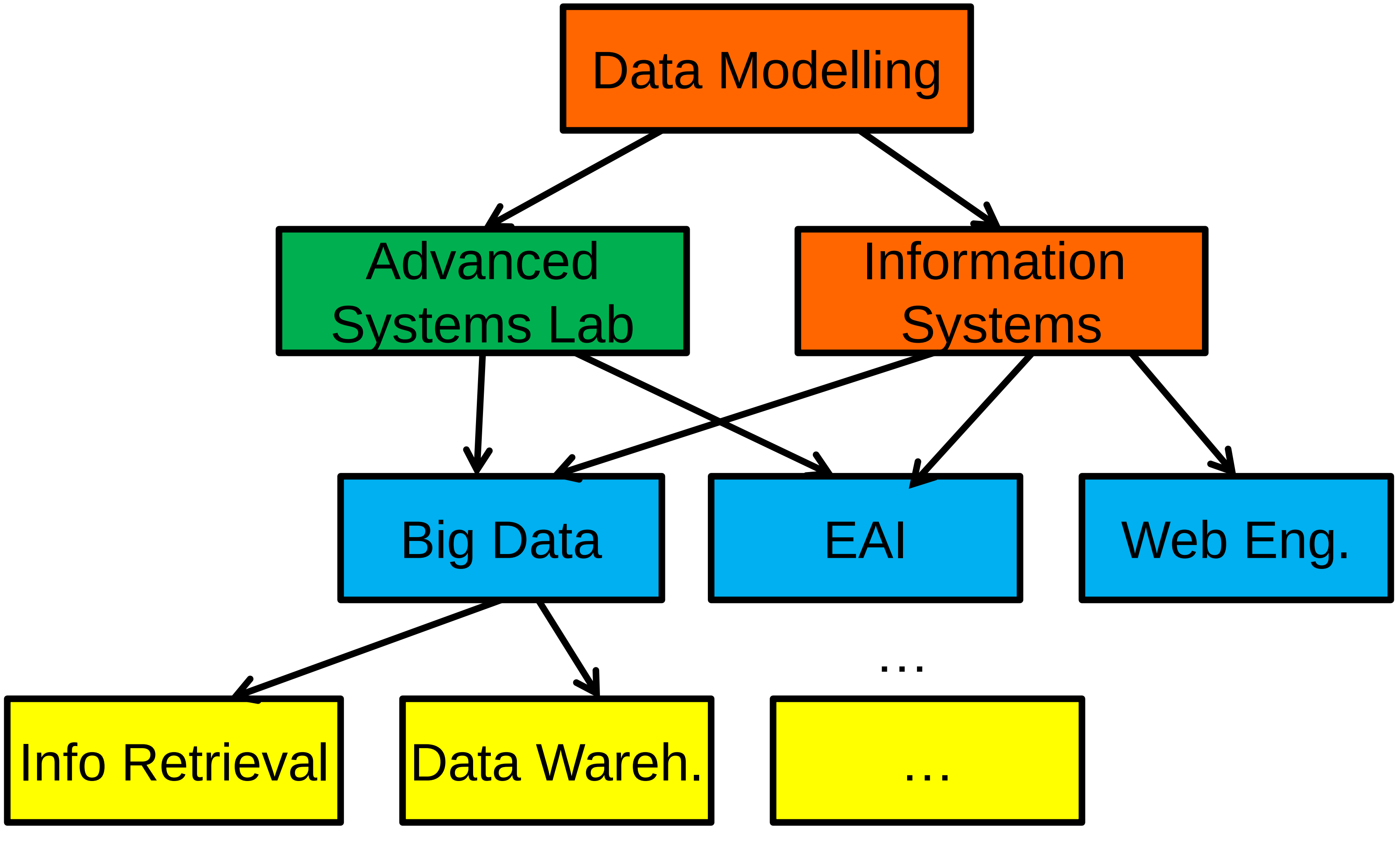
Detailed Schedule

Week No.	Date (Mi)	Topic Lecture	Topic Exercises
1	20.2.2013	Introduction	---
2	27.2.2013	ER, UML	---
3	6.3.2013	Relational Model	ER
4	13.3.2013	SQL I	Start project
5	20.3.2013	Guest Lecture, SQL II	Relational Model
6	27.3.2013	Integrity Constraints	---
7	3.4.2013	---	---
8	10.4.2013	Normal forms I	SQL
9	17.4.2013	Normal forms II	IC, Project: Part I
10	24.4.2013	Query Processing I	Normal forms
11	1.5.2013	(Query Processing II)	Normal forms, Proj.
12	8.5.2013	Transactions	Query Processing
13	15.5.2013	Synchronization	Transactions
14	22.5.2013	Security	Synchronization
15	29.5.2013	Object-relational Databases	End Project: Part 2

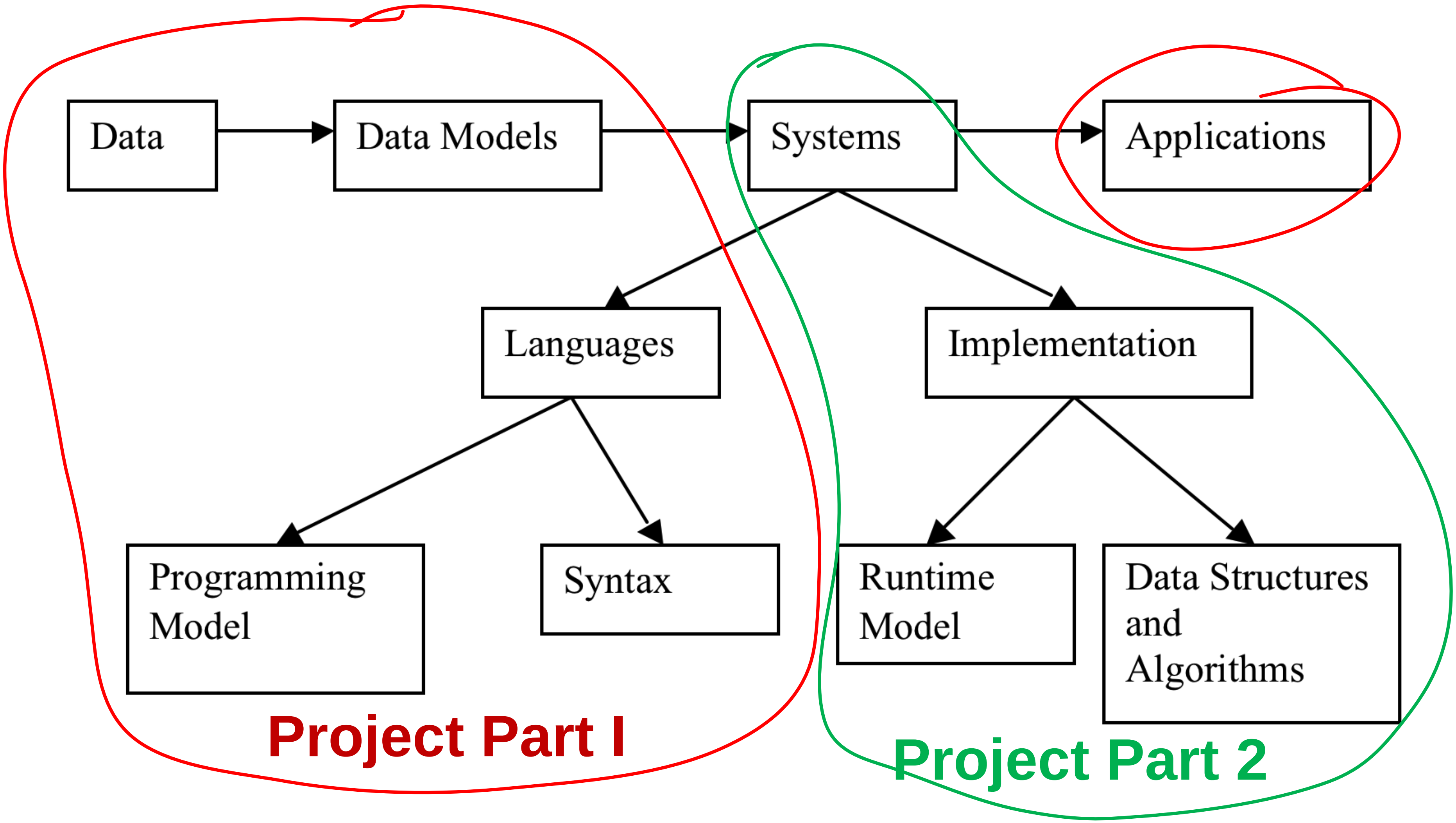
Project Part II

- Starting Point: Existing DB Application (Java + SQL)
  - ideally, your system from Part I of the project
  - otherwise, the student enrollment demo app (+ queries)
- Task
  - Implement a DB library: relational algebra on file system
  - Replace all JDBC statement with calls to your library
- Goal
  - Understand the internals of a database system
  - Understand that there is always an alternative
- Brief glimpse on the „Information Systems“ lecture

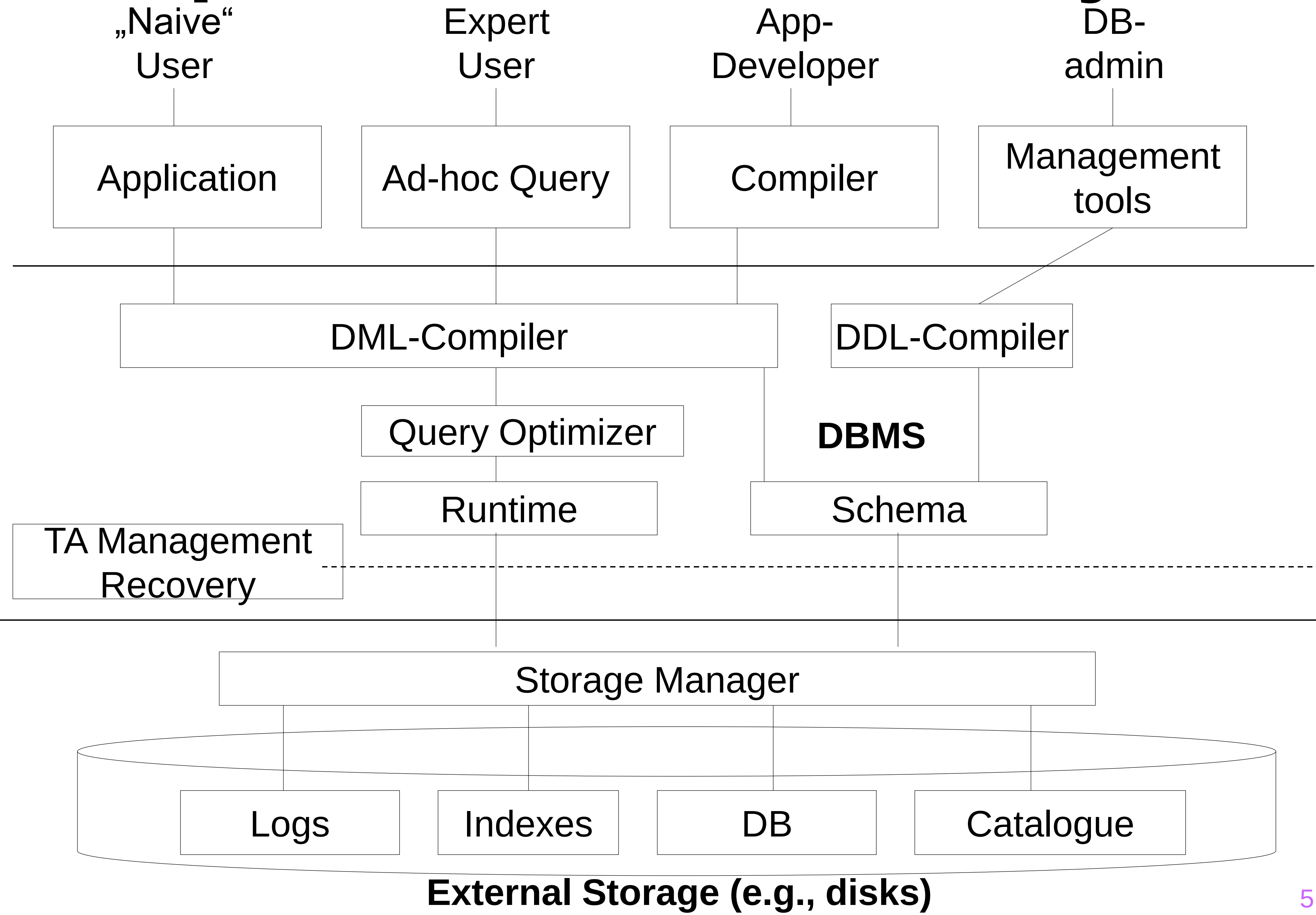
Info Systems Courses: Overview

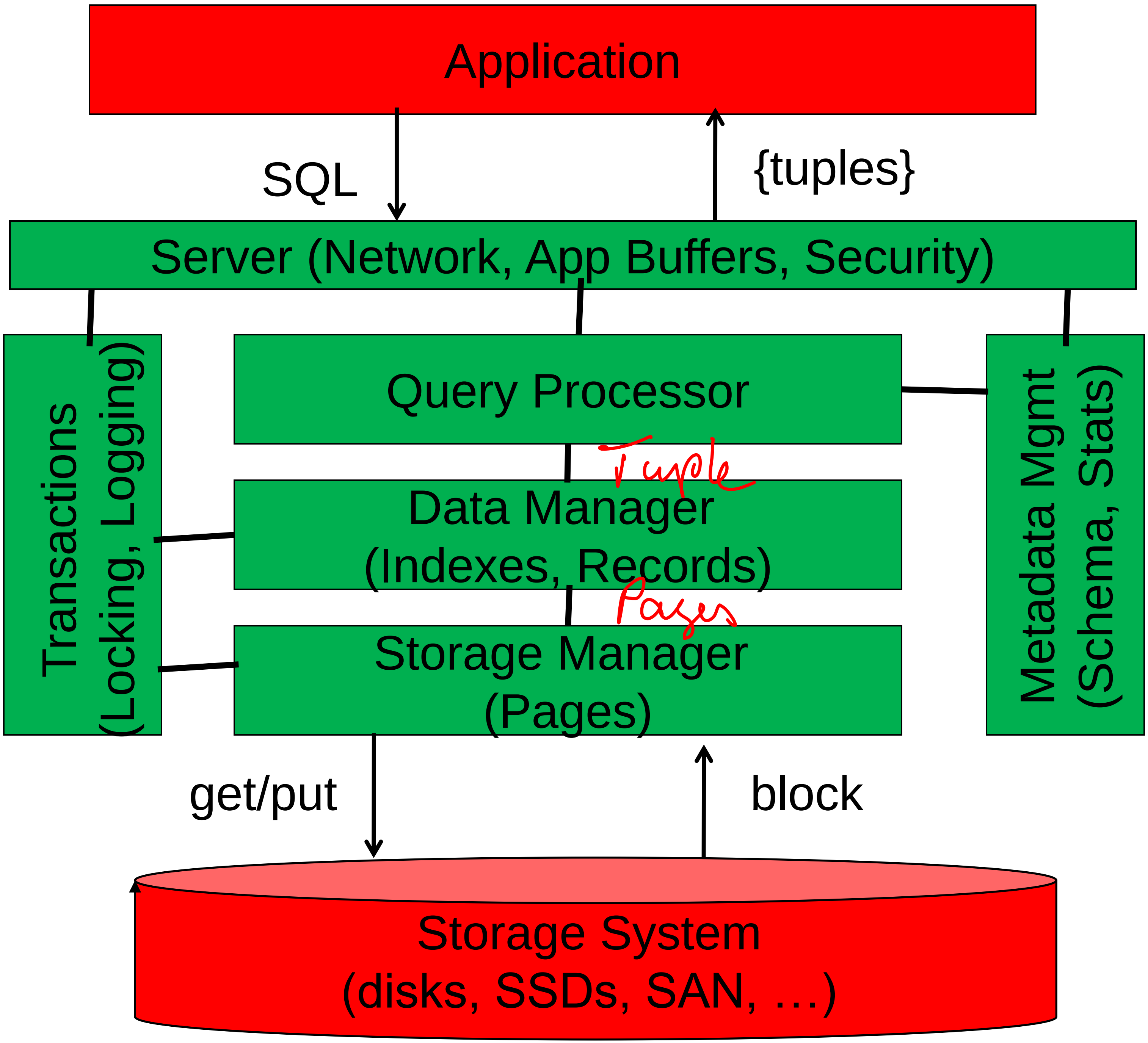


The Data Management Universe



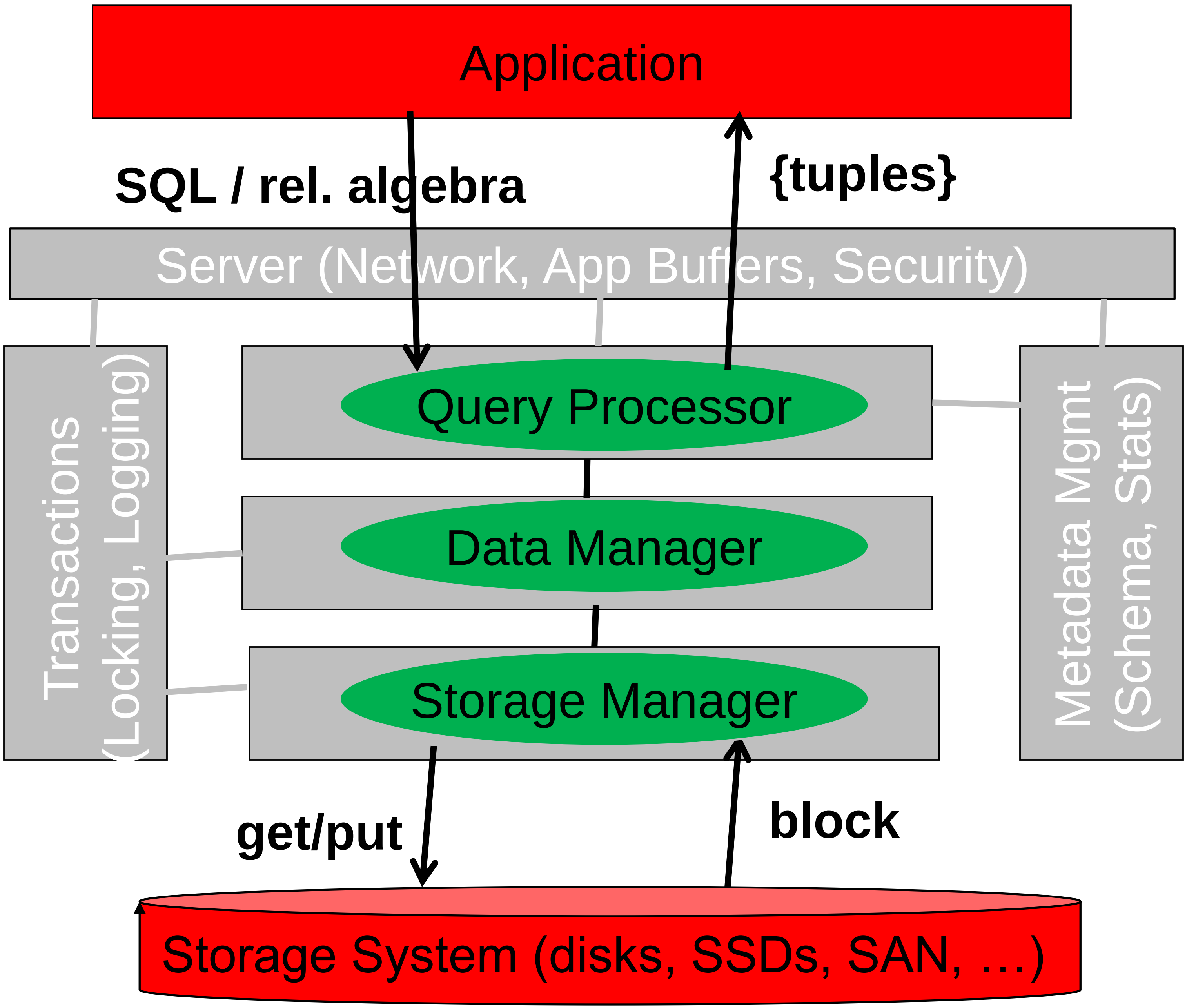
Components of a Database System




Application
SQL
{tuples}
Server (Network, App Buffers, Security)
Transactions (Locking, Logging)
Query Processor
Tuple
Data Manager (Indexes, Records)
Pages
Storage Manager (Pages)
Metadata Mgmt (Schema, Stats)
get/put
block
Storage System (disks, SSDs, SAN, …)

Why use a DBMS?

- Avoid redundancy and inconsistency
  - Query Processor, Transaction Manager, Catalog
- **Rich (declarative) access to the data**
  - **Query Processor**
- Synchronize concurrent data access
  - Transaction Manager
- Recovery after system failures
  - Transaction Manager, Storage Layer
- Security and privacy
  - Server, Catalog



Project Part II

What does a Database System do?

- Input: SQL statement
- Output: {tuples}

1. *Translate SQL into a set of get/put req. to backend storage*
2. *Extract, process, transform tuples from blocks*

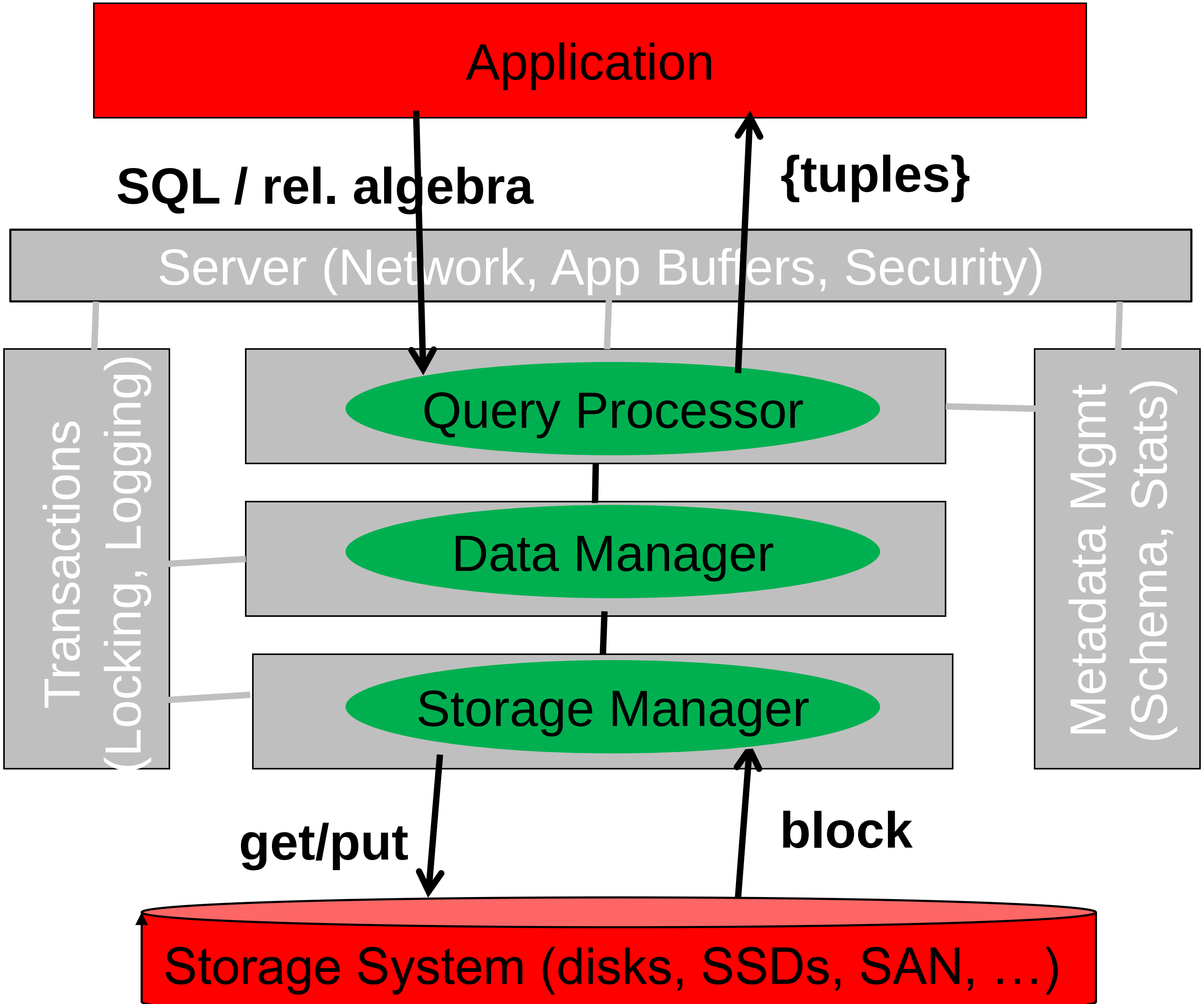
- Tons of optimizations
  - Efficient algorithms for SQL operators (hashing, sorting)
  - Layout of data on backend storage (clustering, free space)
  - Ordering of operators (small intermediate results)
  - Semantic rewritings of queries
  - Buffer management and caching
  - Parallel execution and concurrency
  - Outsmart the OS
  - Partitioning and Replication in distributed system
  - Indexing and Materialization
  - Load and admission control
- + Security + Durability + Concurrency Control + Tools

Database Optimizations

- Query Processor (based on statistics)
  - Efficient algorithms for SQL operators (hashing, sorting)
  - Ordering of operators (small intermediate results)
  - Semantic rewritings of queries
  - Parallel execution and concurrency
- Storage Manager
  - Load and admission control
  - Layout of data on backend storage (clustering, free space)
  - Buffer management and caching
  - Outsmart the OS
- Transaction Manager
  - Load and admission control
- Tools (based on statistics)
  - Partitioning and Replication in distributed system
  - Indexing and Materialization

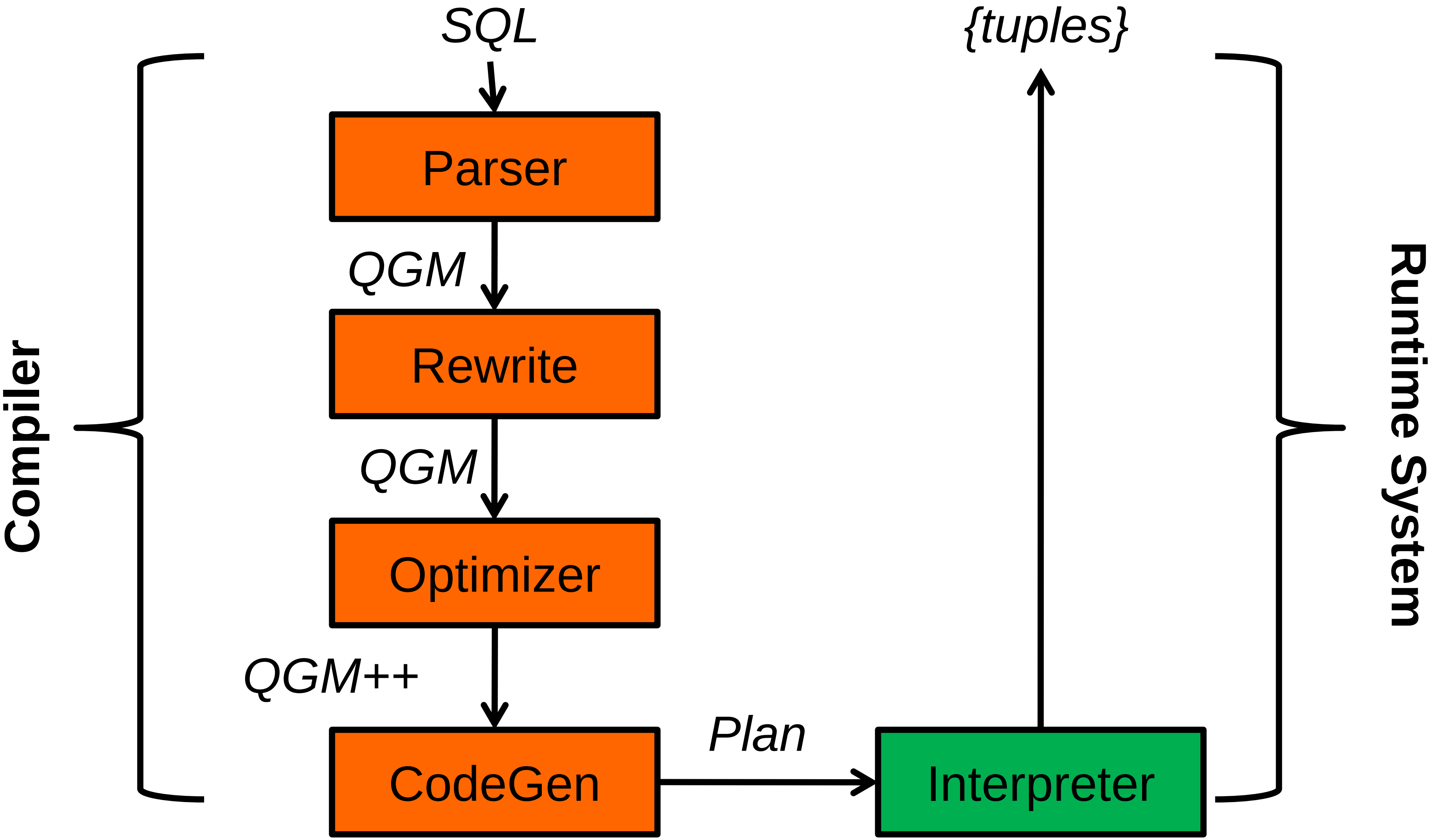
DBMS vs. OS Optimizations

- Many DBMS tasks are also carried out by OS
  - Load control
  - Buffer management
  - Access to external storage
  - Scheduling of processes
  - ...
- What is the difference?
  - DBMS has intimate knowledge of workload
  - DBMS can predict and shape access pattern of a query
  - DBMS knows the mix of queries (all pre-compiled)
  - DBMS knows the contention between queries
  - OS does generic optimizations
- ***Problem: OS overrides DBMS optimizations!***



Application
SQL / rel. algebra
{tuples}
Server (Network, App Buffers, Security)
Transactions
(Locking, Logging)
Query Processor
Data Manager
Storage Manager
Metadata Mgmt
(Schema, Stats)
get/put
block
Storage System (disks, SSDs, SAN, …)

Query Processor



SQL -> Relational Algebra

SQL

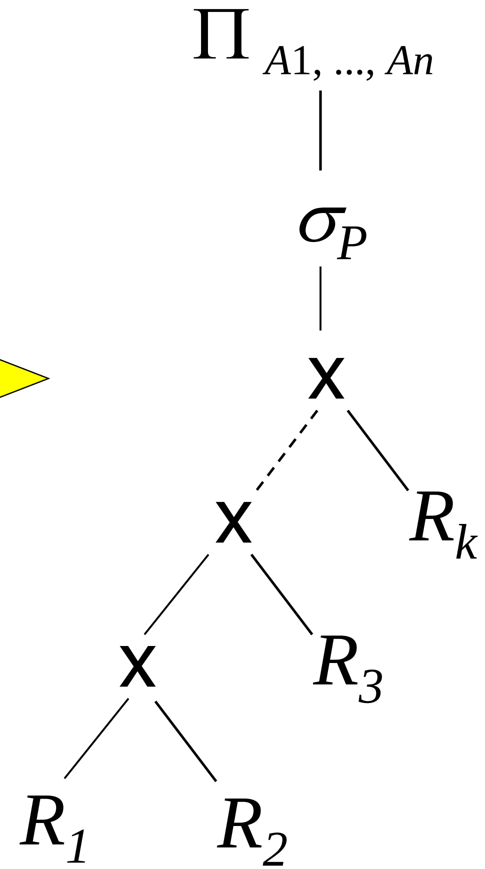
Relational Algebra

$$\Pi_{A1, \ldots, An}(\sigma_P (R_1 \times \ldots \times R_k))$$

select $A_1, \ldots, A_n$

from $R_1, \ldots, R_k$

where P;



Runtime System

- Three approaches
  - A. Compile query into machine code
  - B. Compile query into relational algebra and interpret that
  - C. Hybrid: e.g., compile predicates into machine code

- What to do?
  - A: better performance
  - B: easier debugging, better portability
  - Project: use Approach B

- Query Interpreter
  - provide implementation for each algebra operator
  - define interface between operators

Algorithms for Relational Algebra

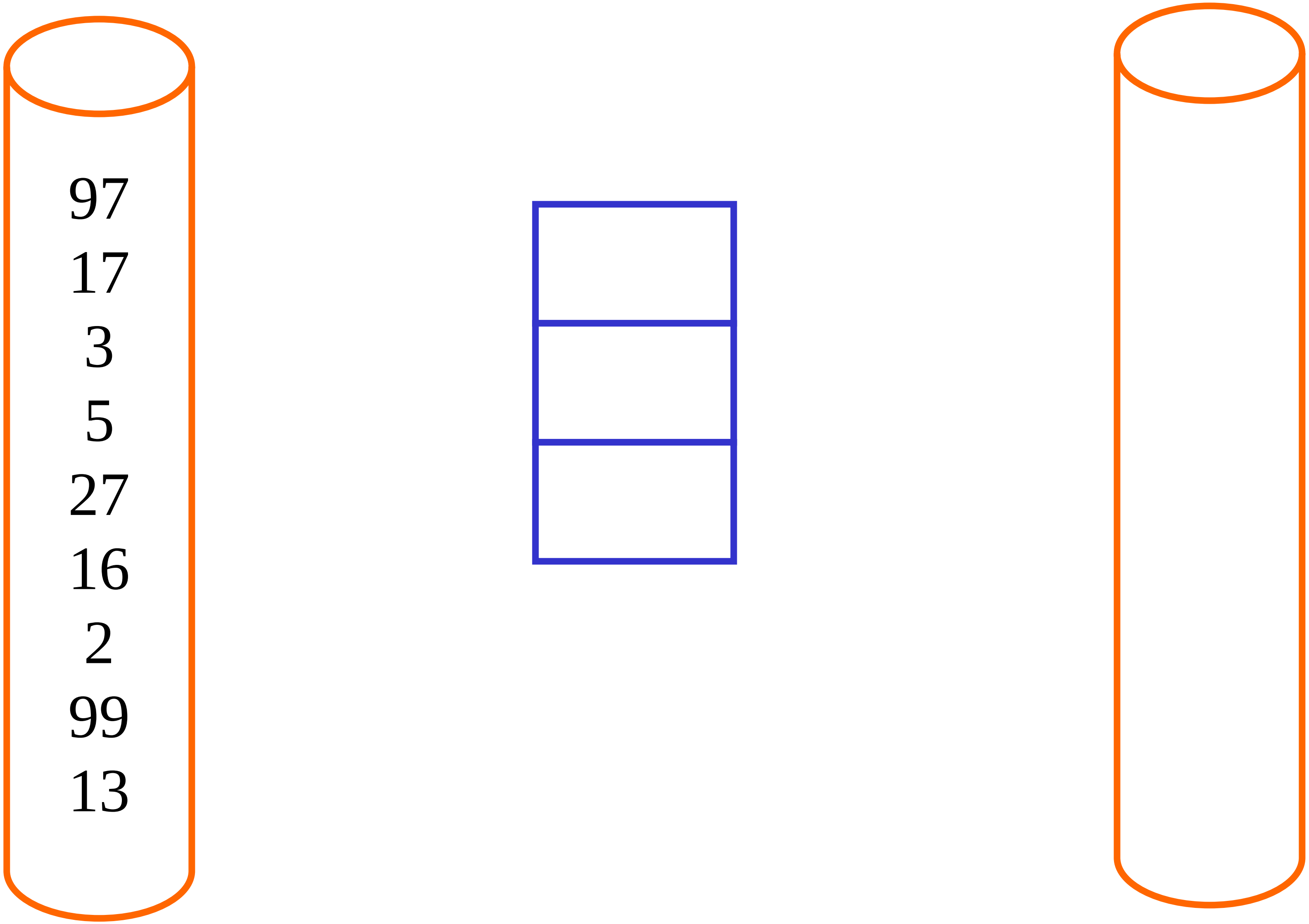
- Table Access
  - scan (load each page at a time)
  - index scan (if index available)
- Sorting
  - Two-phase external sorting
- Joins
  - (Block) nested-loops
  - Index nested-loops
  - Sort-Merge
  - Hashing (many variants)
- Group-by (~ self-join)
  - Sorting
  - Hashing

Two-phase External Sorting

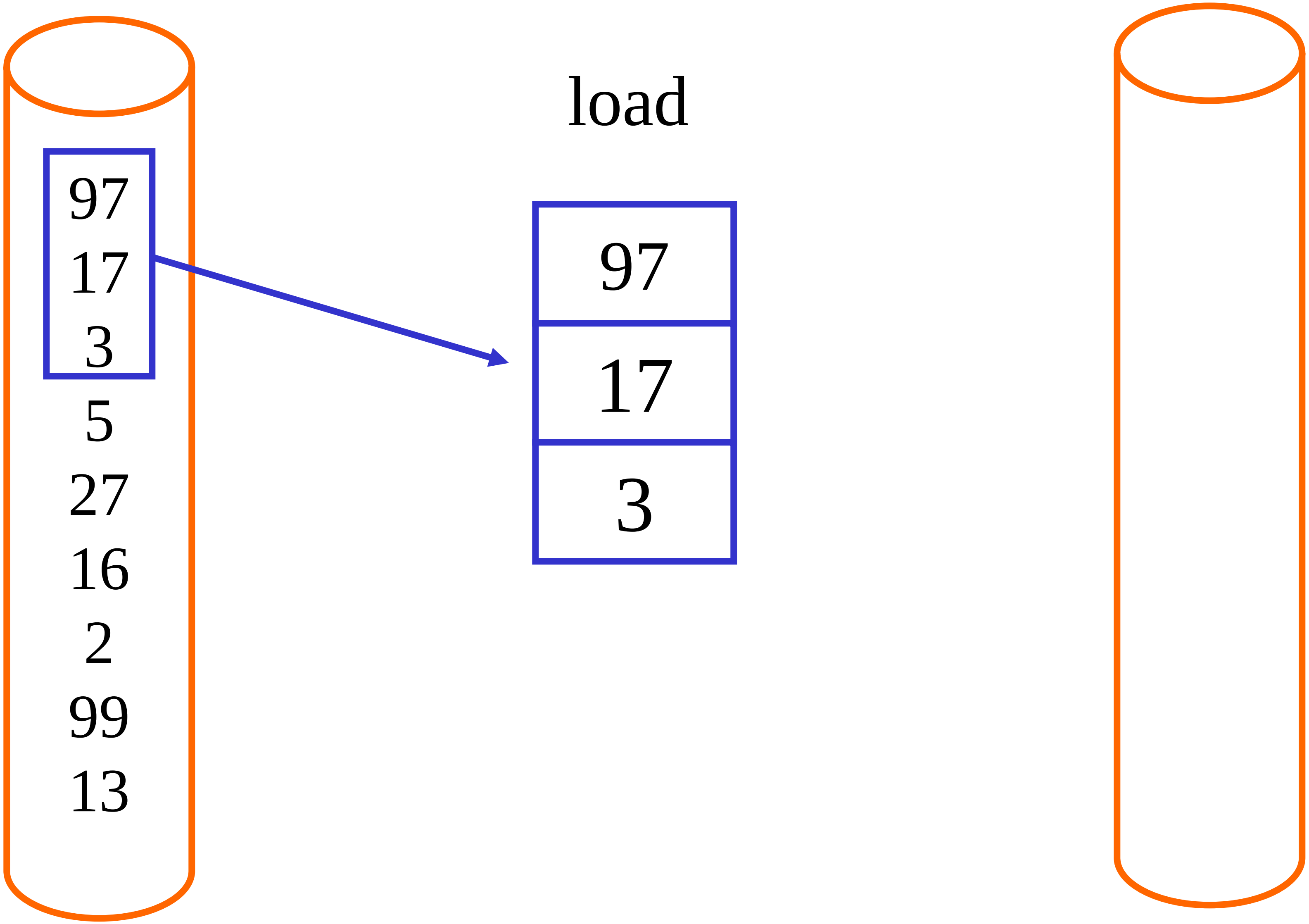
- Phase I: Create Runs
  1. Load allocated buffer space with tuples
  2. Sort tuples in buffer pool
  3. Write sorted tuples (run) to disk
  4. Goto Step 1 (create next run) until all tuples processed
- Phase II: Merge Runs
  - Use priority heap to merge tuples from runs

- Special cases
  - buffer >= N: no merge needed
  - buffer < sqrt(N): multiple merge phases necessary
  - (N size of the input in pages)

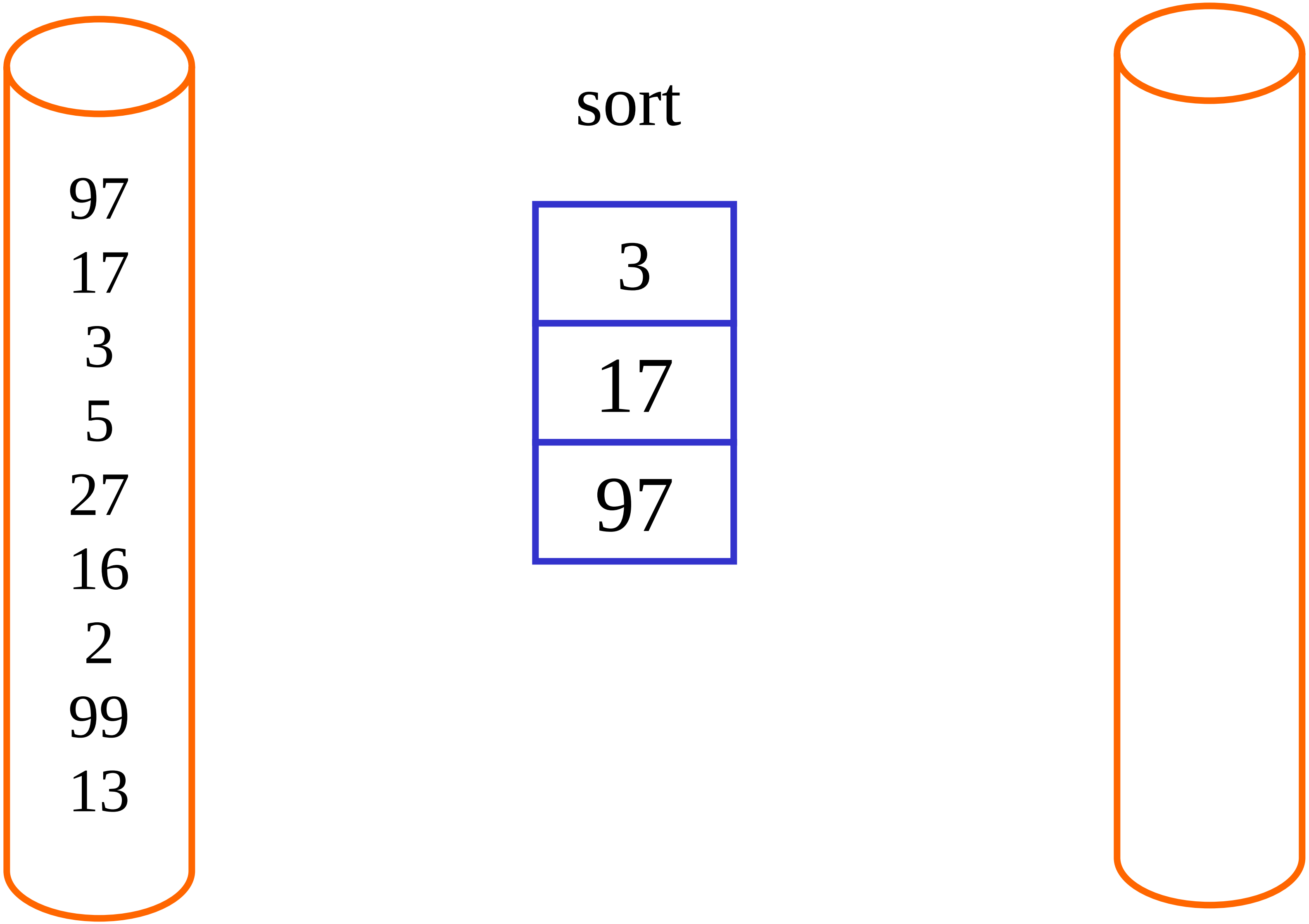
External Sort



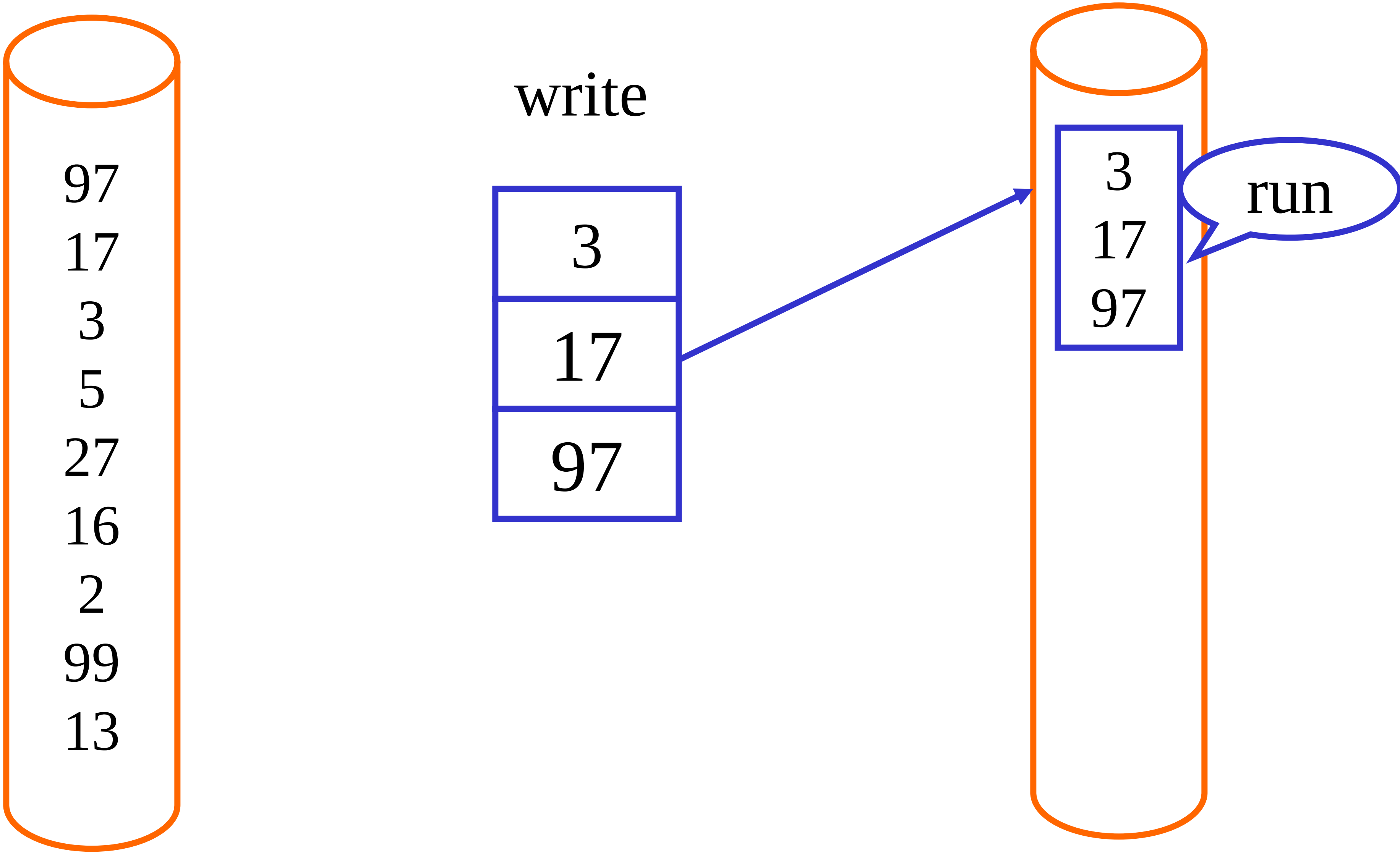
External Sort



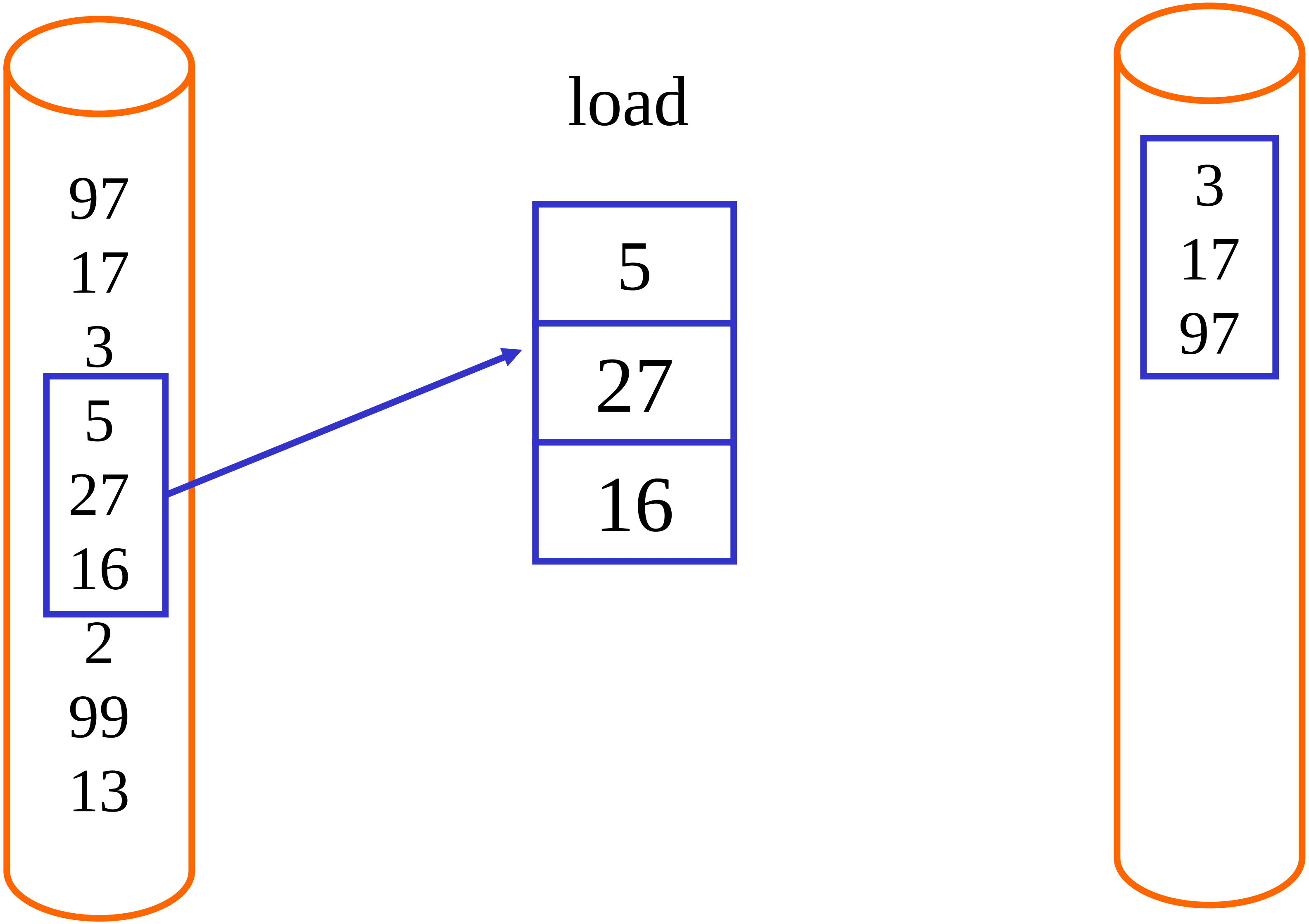
External Sort



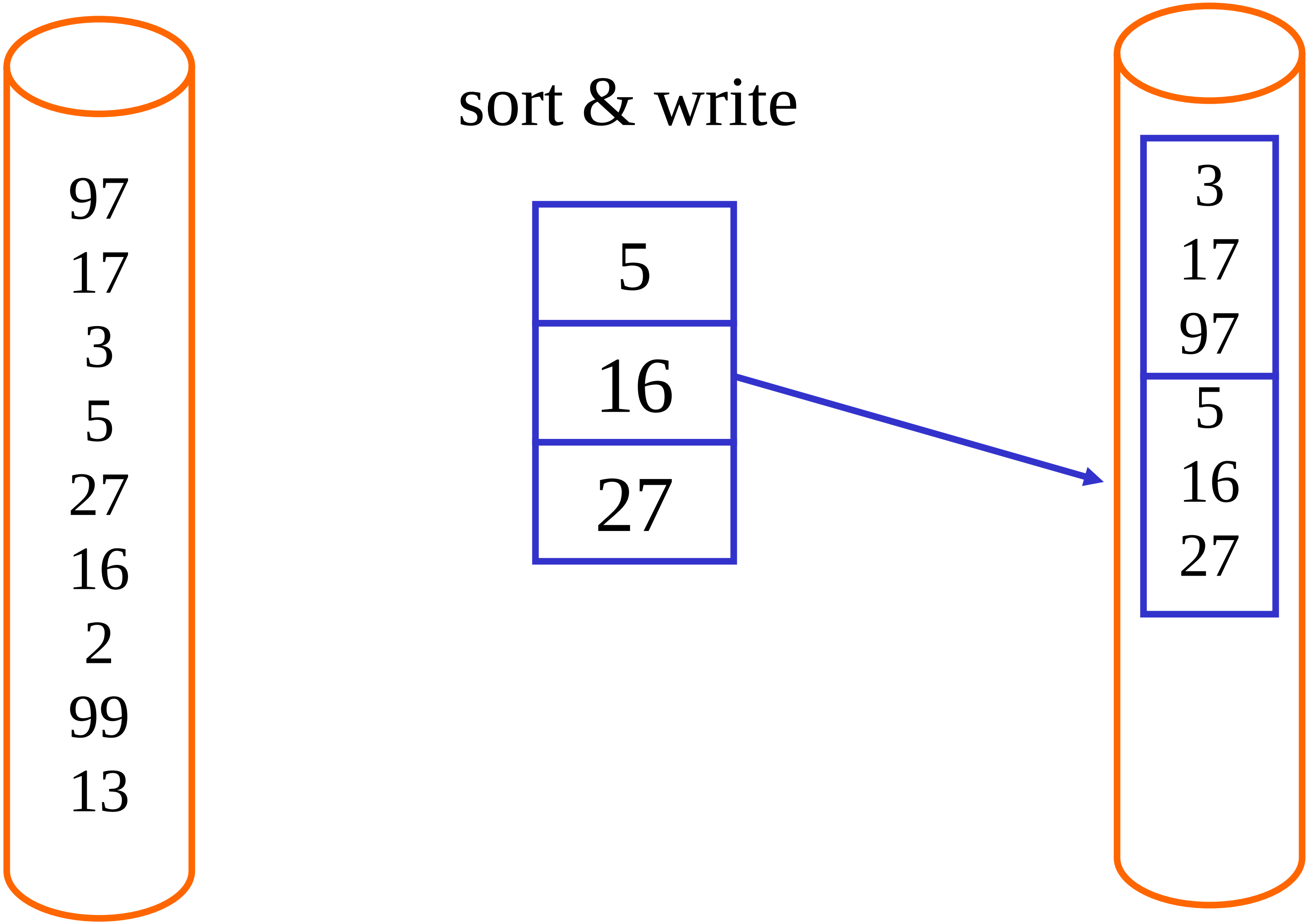
External Sort



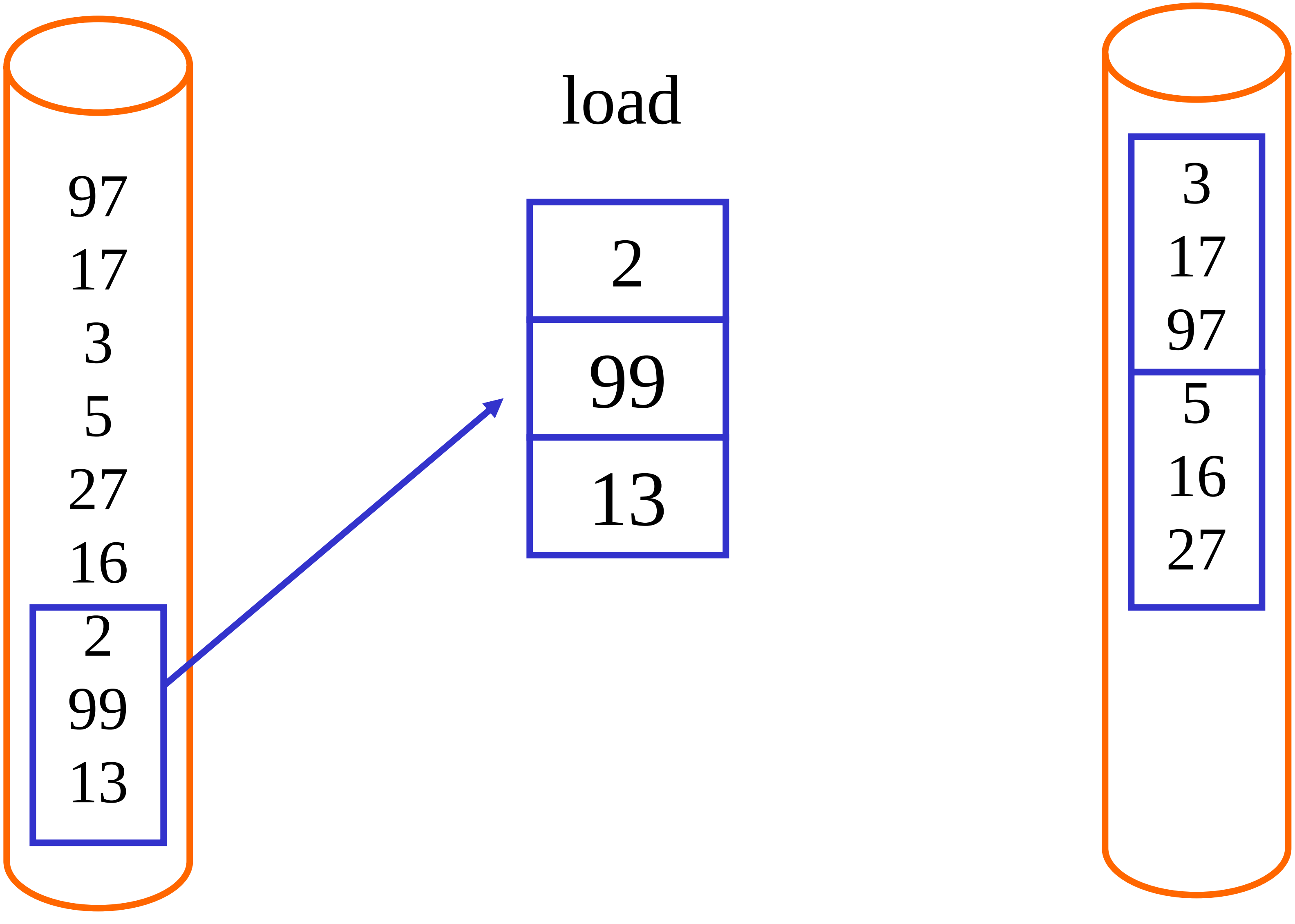
External Sort



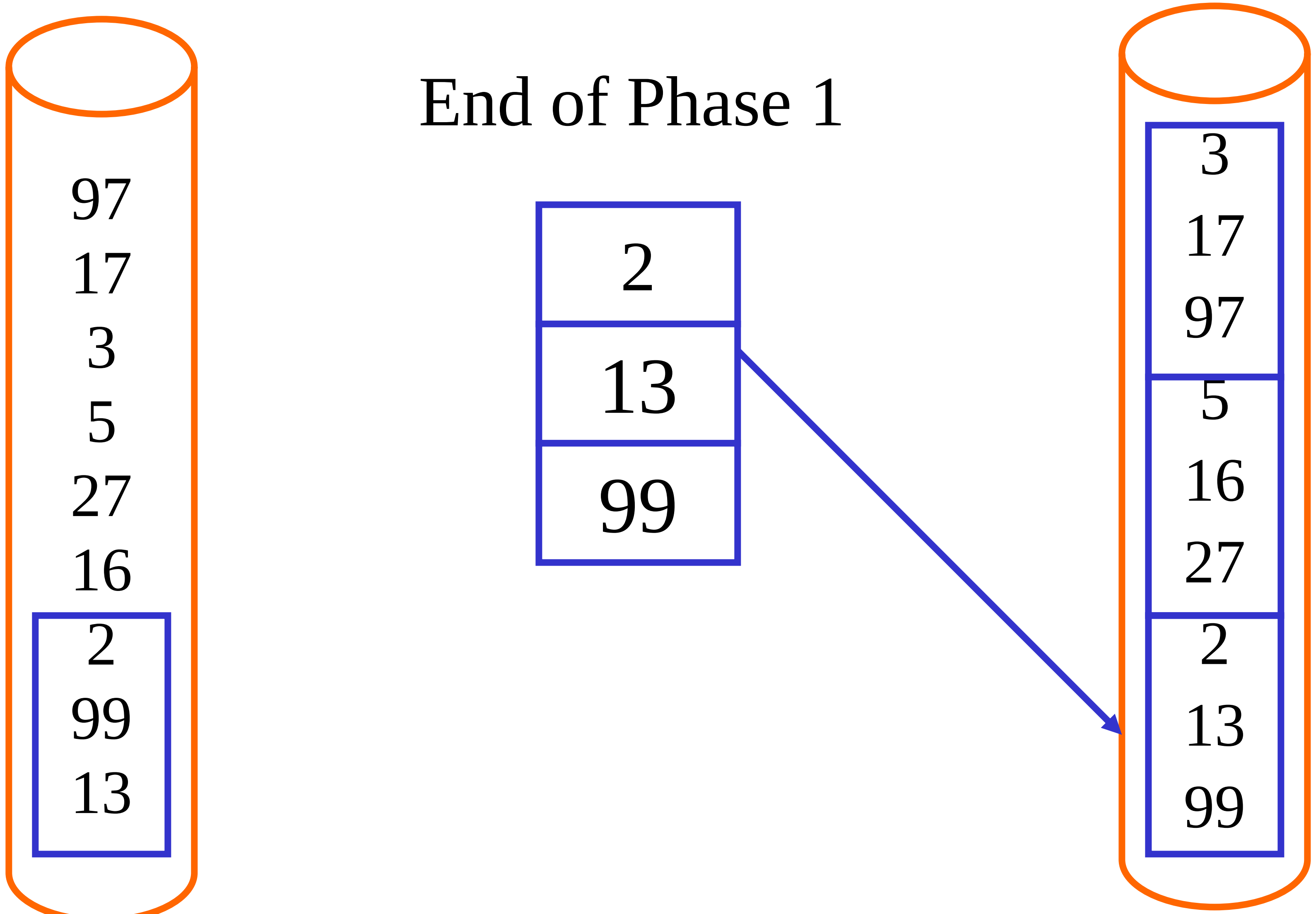
External Sort



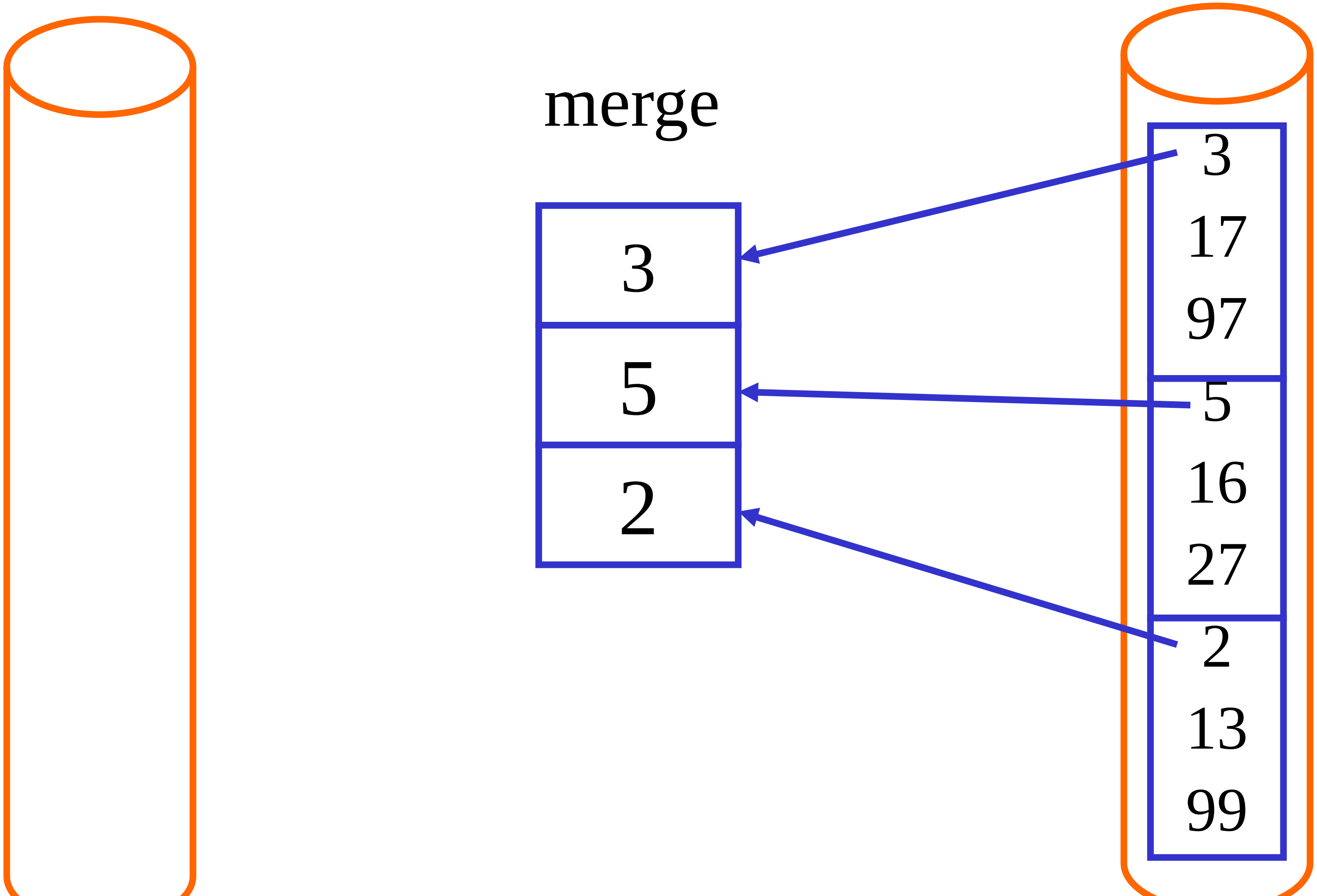
External Sort



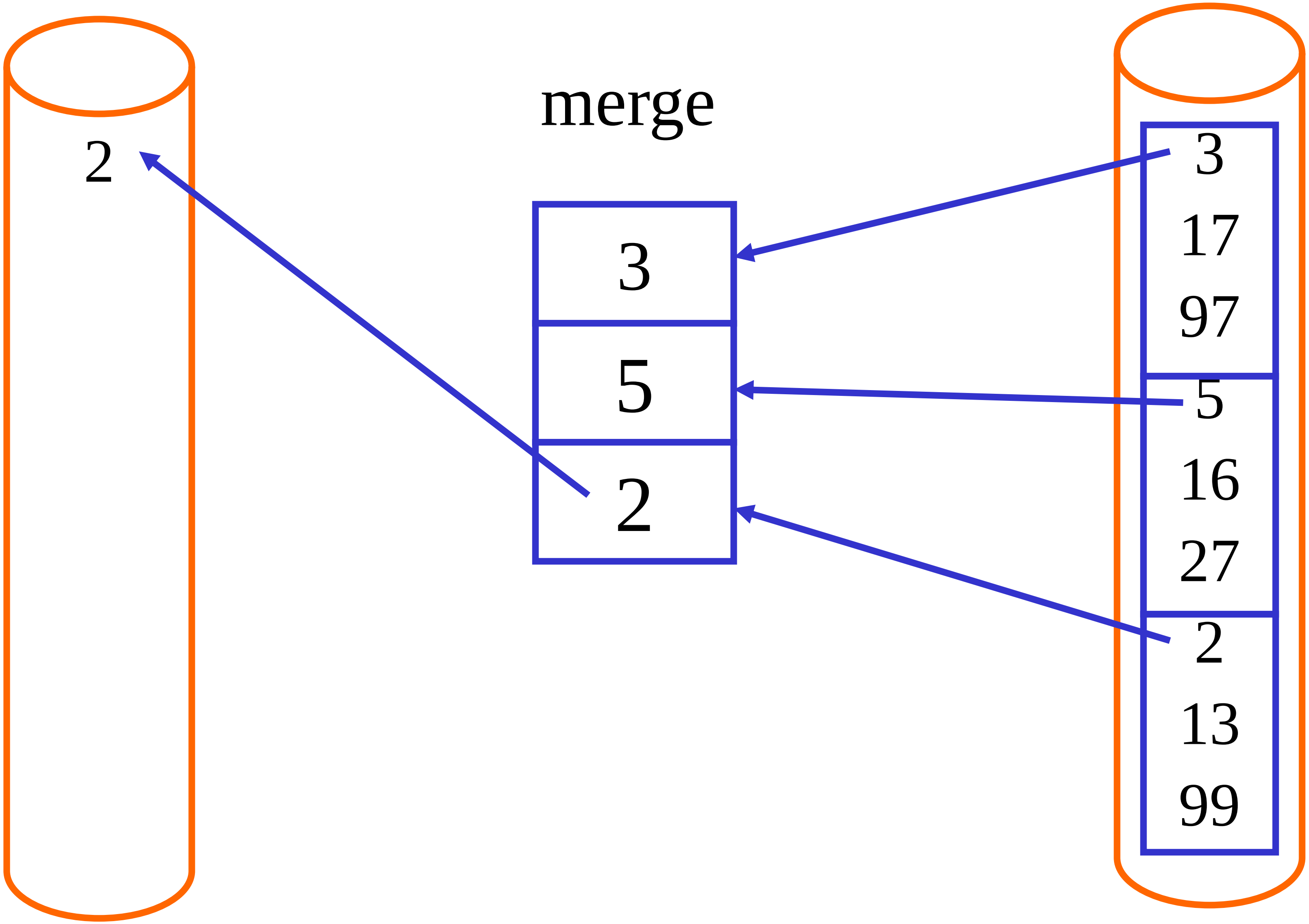
External Sort



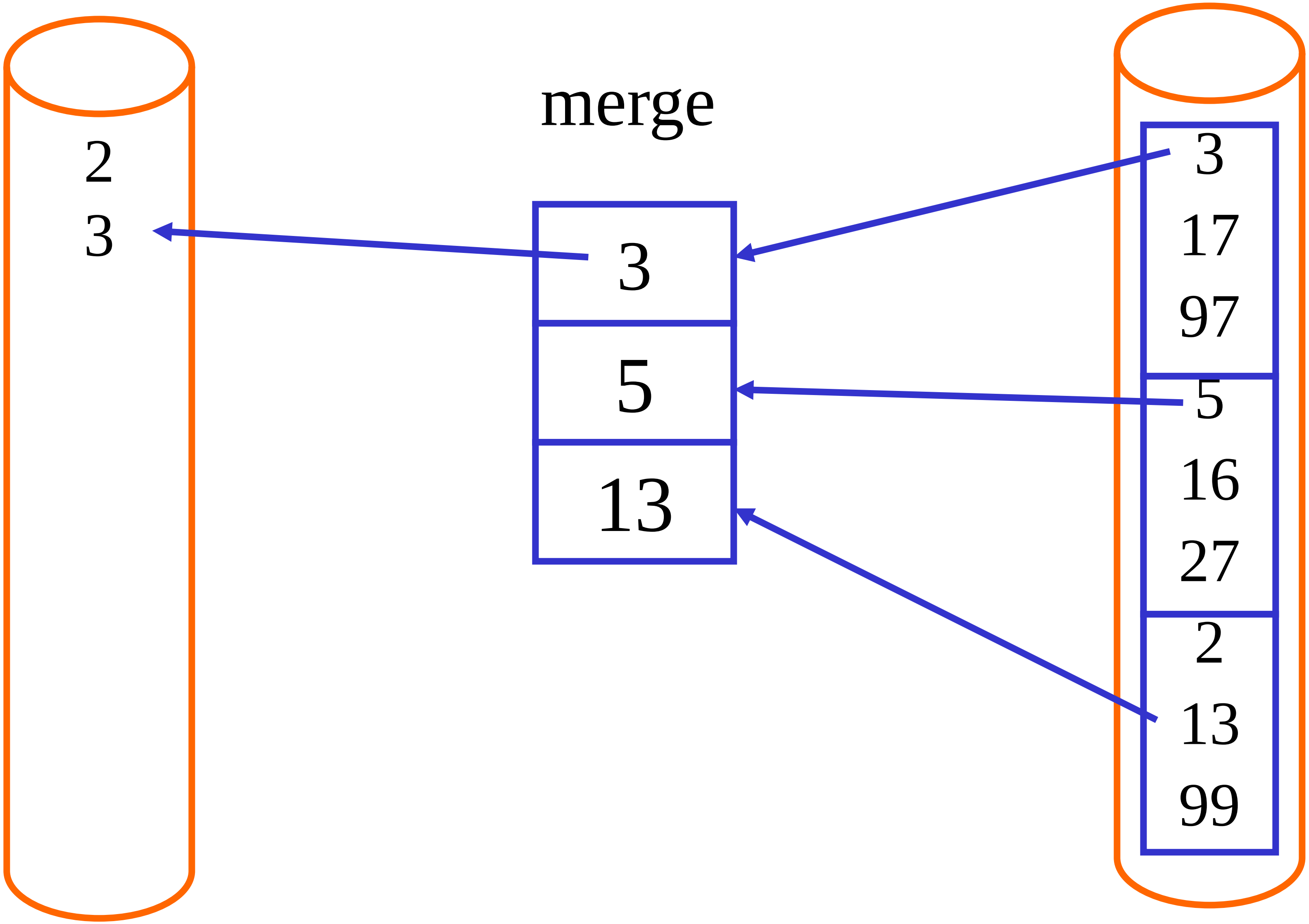
External Sort



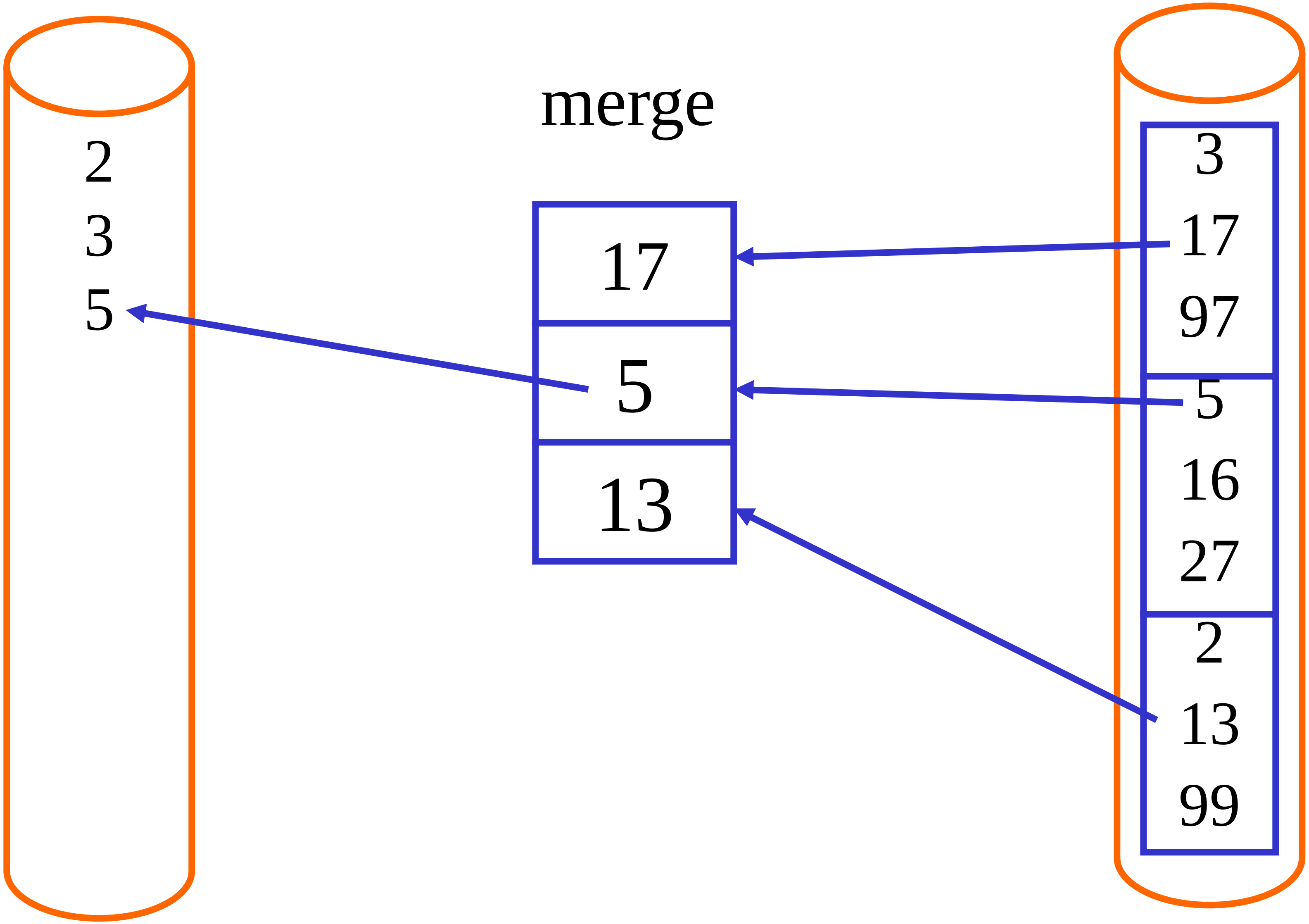
External Sort



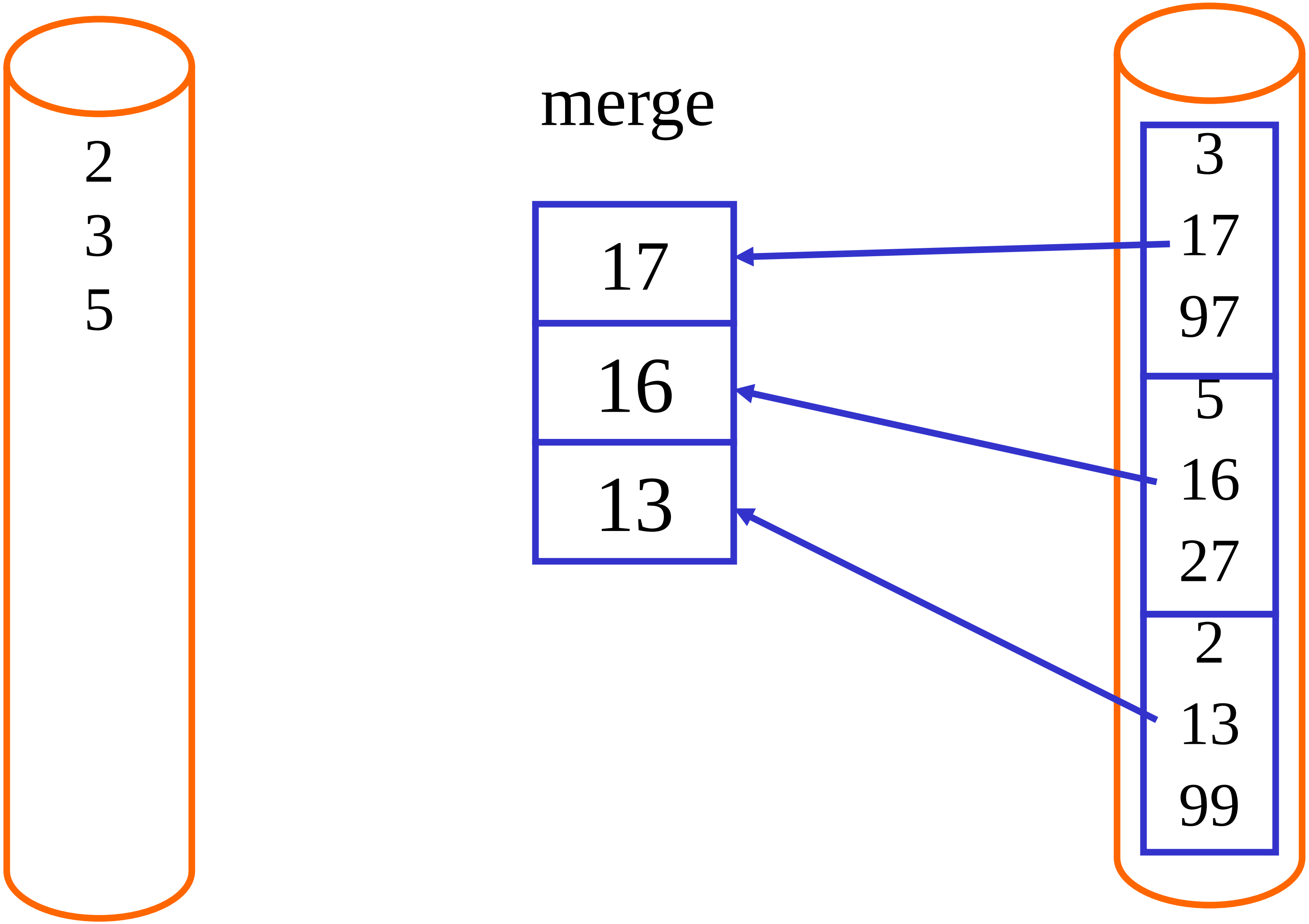
External Sort



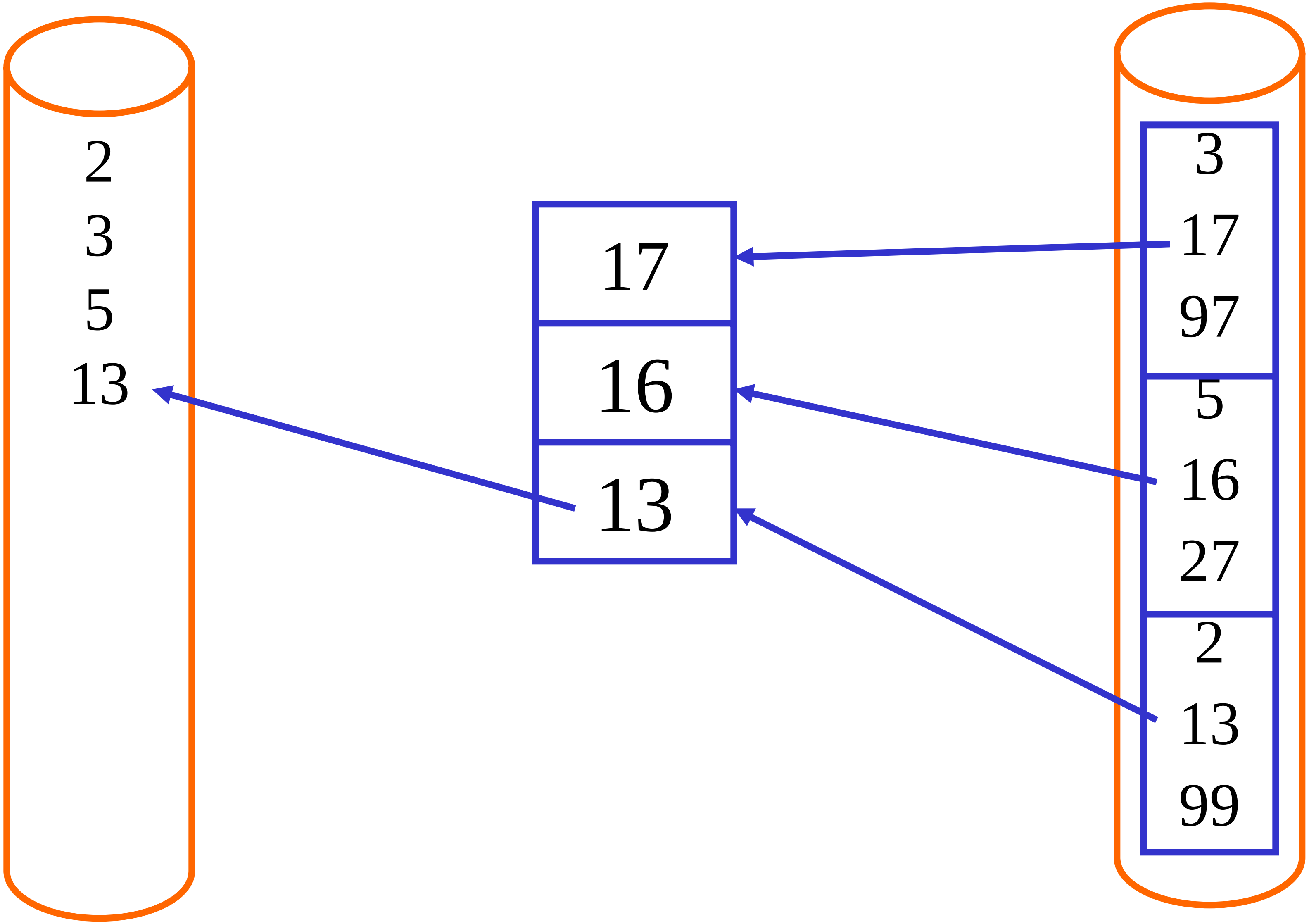
External Sort



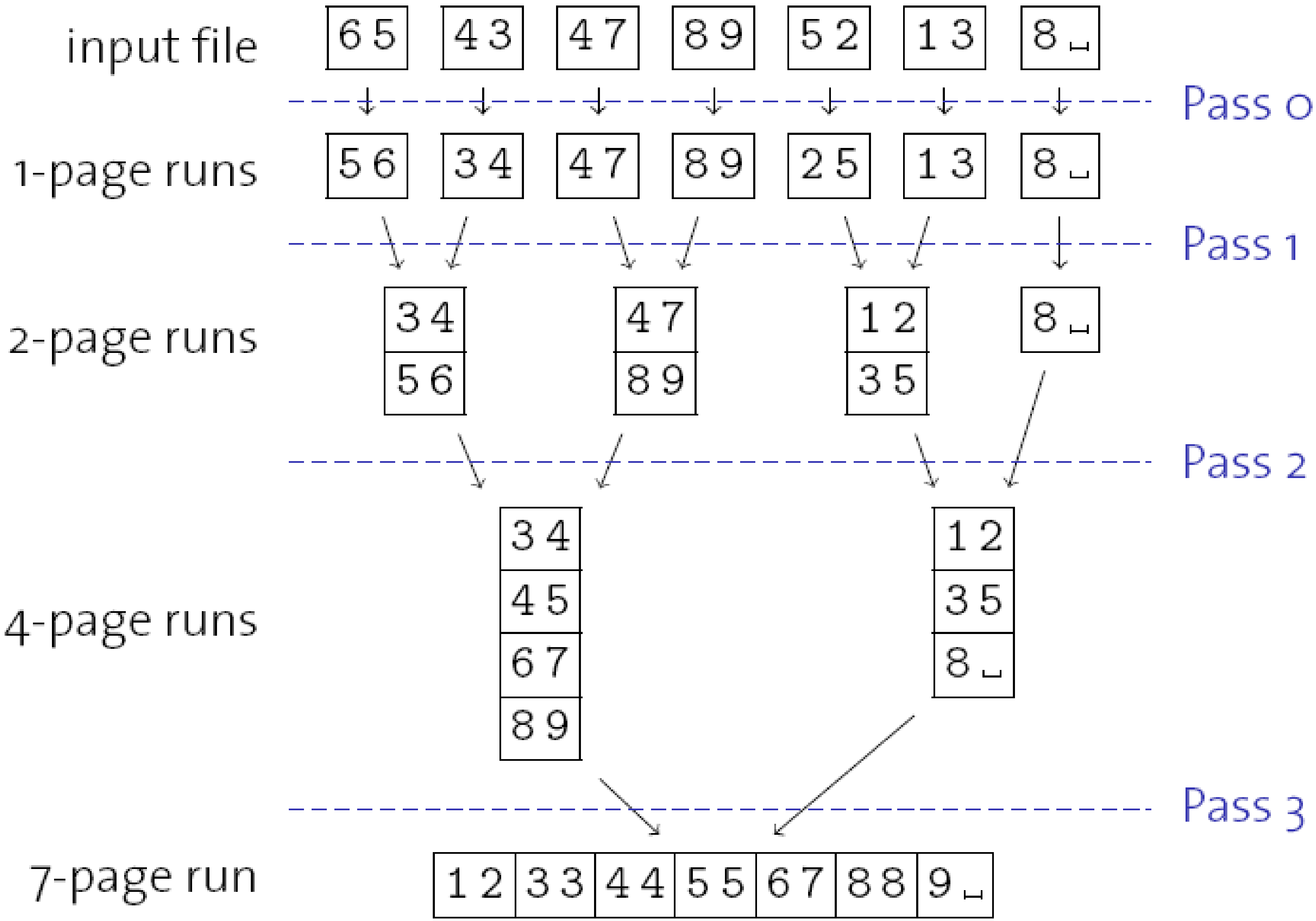
External Sort



External Sort



Multi-way Merge (N = 7; M = 2)



Analysis

- N: size of table in pages
- M: size of (available) main memory in pages
- IO Cost
  - $\mathcal{O}(N)$: if M >= sqrt(N)
    - 2 * N: if M >= N
    - 4 * N: if N > M >= sqrt(N)
  - $\mathcal{O}(N \log_M N)$: if M < sqrt(N)
    - Base of logarithm: in $\mathcal{O}$ notation not relevant, but constants matter
- CPU Cost (M >= sqrt(N))
  - Phase 1 (create N/M runs of length M): $\mathcal{O}(N * \log_2 M)$
  - Phase 2 (merge N tuples with heap): $O(N * \log_2 N/M)$
  - Exercise: Do CPU cost increase/decrease with M?

Sorting Summary

- Complexity: N * log(N)   theory is right, but
  - DB people care about CPU and IO complexity
  - Constants matter!
  - Buffer allocation matters! Many concurrent queries?
  - More main memory can hurt performance!
- Main memory is large. Do two-way sort because…
  - Parallelize sorting on different machines
  - Or many concurrent sorts on same machine
  - But more than 2-ways very rare in practice
- Knuth suggests Replacement Selection
  - Increases length of runs
  - But, higher constant for CPU usage
  - Typically, not used in practice

(Grace) Hash Join

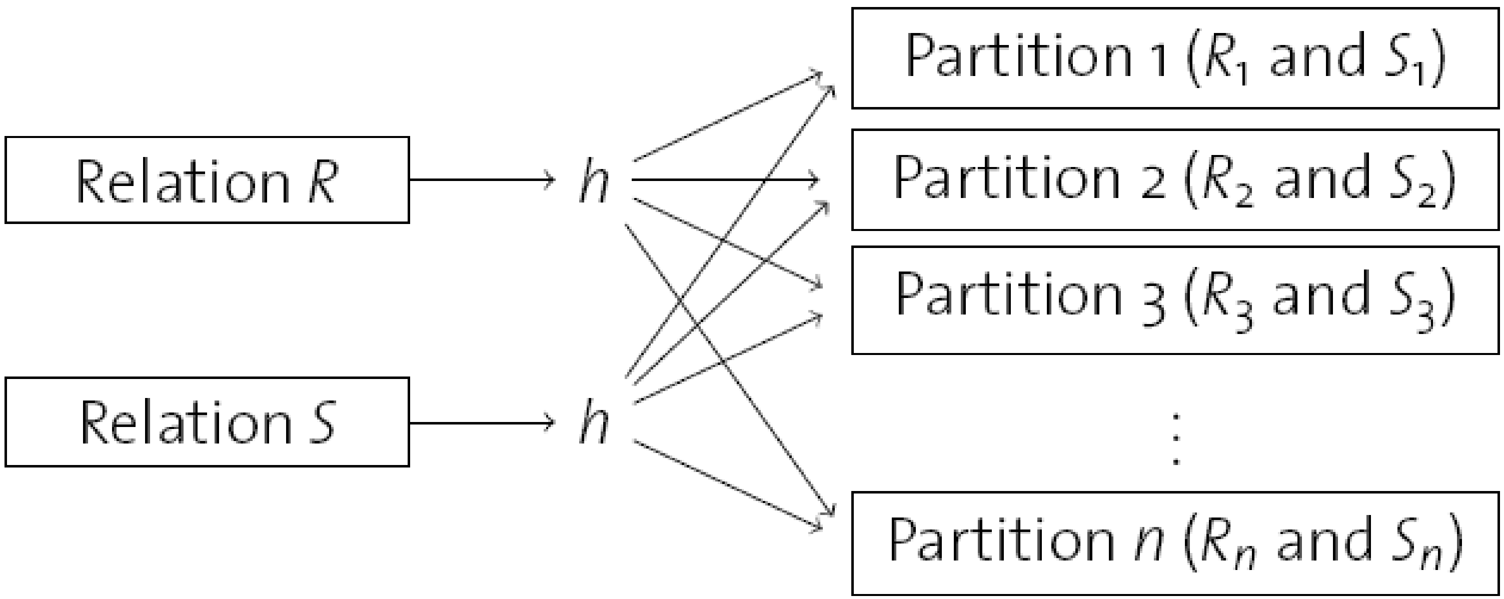
Function: `hash_join` $(R, S, \alpha = \beta)$

foreach record $r \in R$ **do**
    append r to partition $R_{h(r.\alpha)}$

foreach record $s \in S$ **do**
    append s to partition $S_{h(s.\beta)}$

foreach partition $i \in 1, \ldots, n$ **do**
    build hash table H for R_i, using hash function h';
    foreach block in S_i **do**
        foreach record s in current S_i-block **do**
            probe H and append matching tuples to result ;

Grace Hash Join



$$R_i \bowtie S_j = \varnothing \text{ for all } i \neq j$$

Sorting vs. Hashing

- Both techniques can be used for joins, group-by, …
  - binary and unary matching problems
- Same asymptotic complexity: $\mathcal{O}(N \log N)$
  - In both IO and CPU
  - Hashing has lower constants for CPU complexity
  - IO behavior is almost identical
- Merging (Sort) vs. Partitioning (Hash)
  - Merging done *afterwards;* Partitioning done *before*
  - Partitioning depends on good statistics to get right
- Sorting more robust. Hashing better in average case!

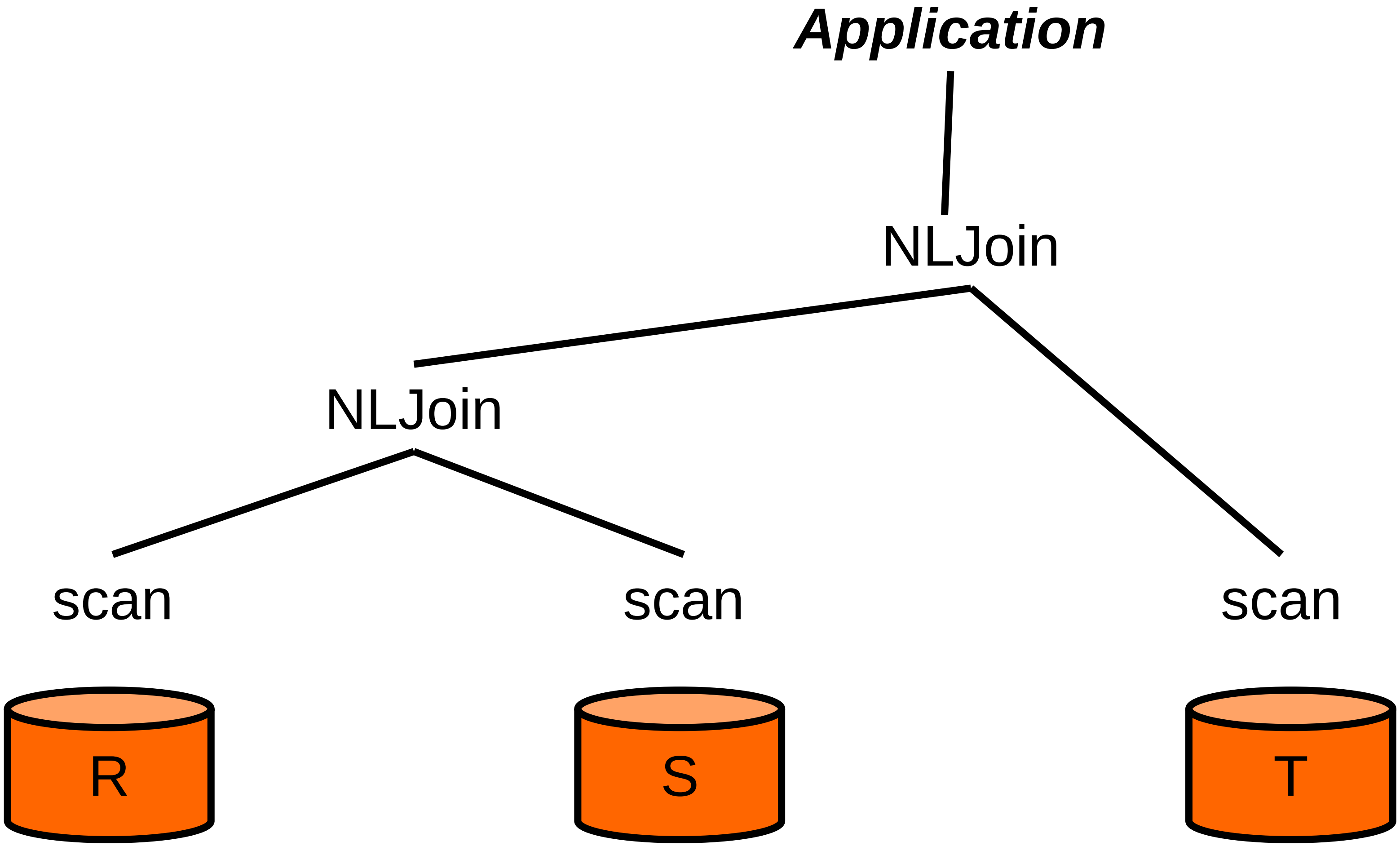
Iterator Model

- Plan contains many operators
  - Implement each operator indepently
  - Define generic interface for each operator
  - Each operator implemented by an *iterator*

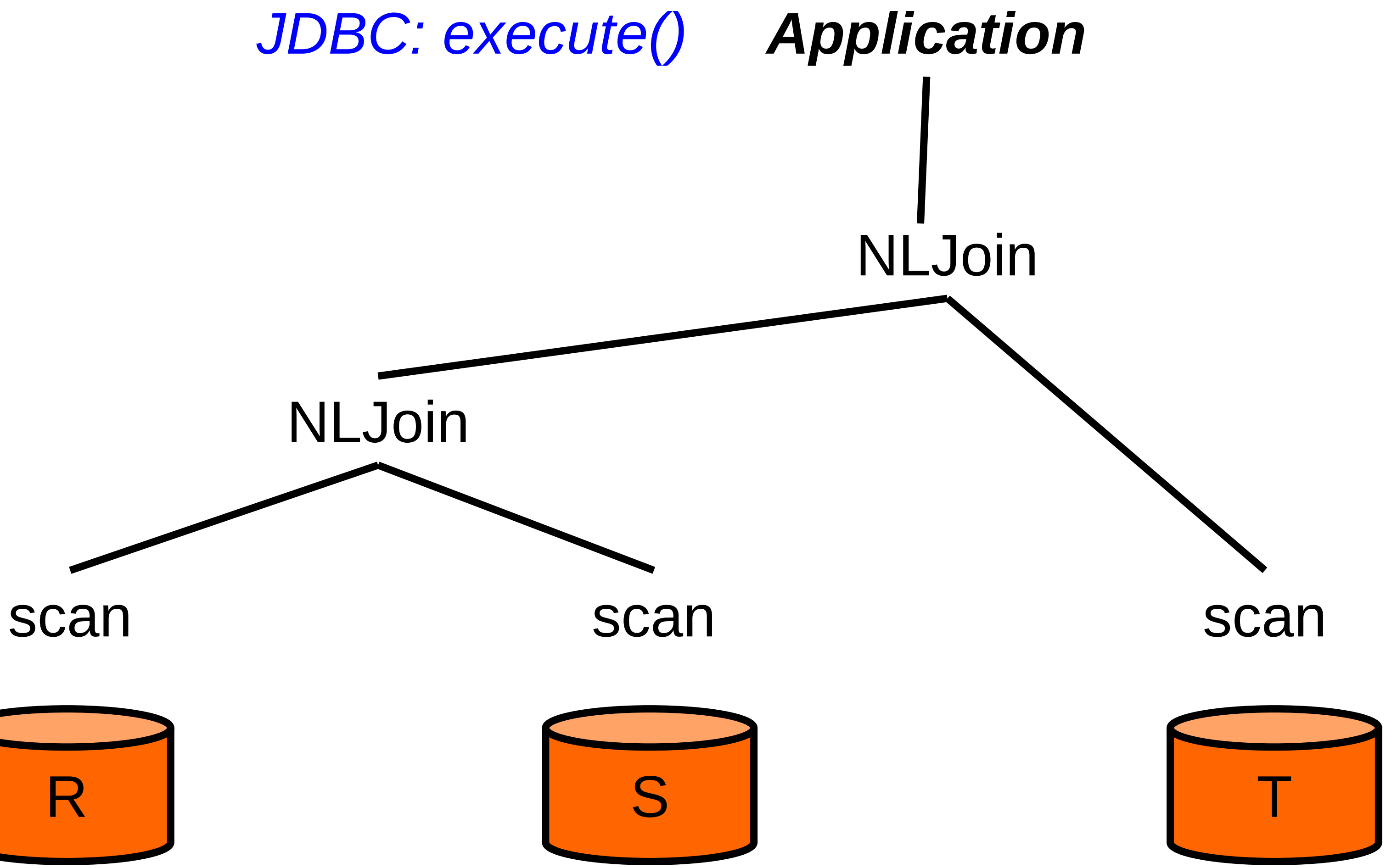
- Three methods implemented by each iterator
  - open(): initialize the internal state (e.g., allocate buffer)
  - char* next(): produce the next result tuple
  - close(): clean-up (e.g., release buffer)

- N.B. Modern DBMS use a Vector Model
  - next() returns a set of tuples
  - Why is that better?

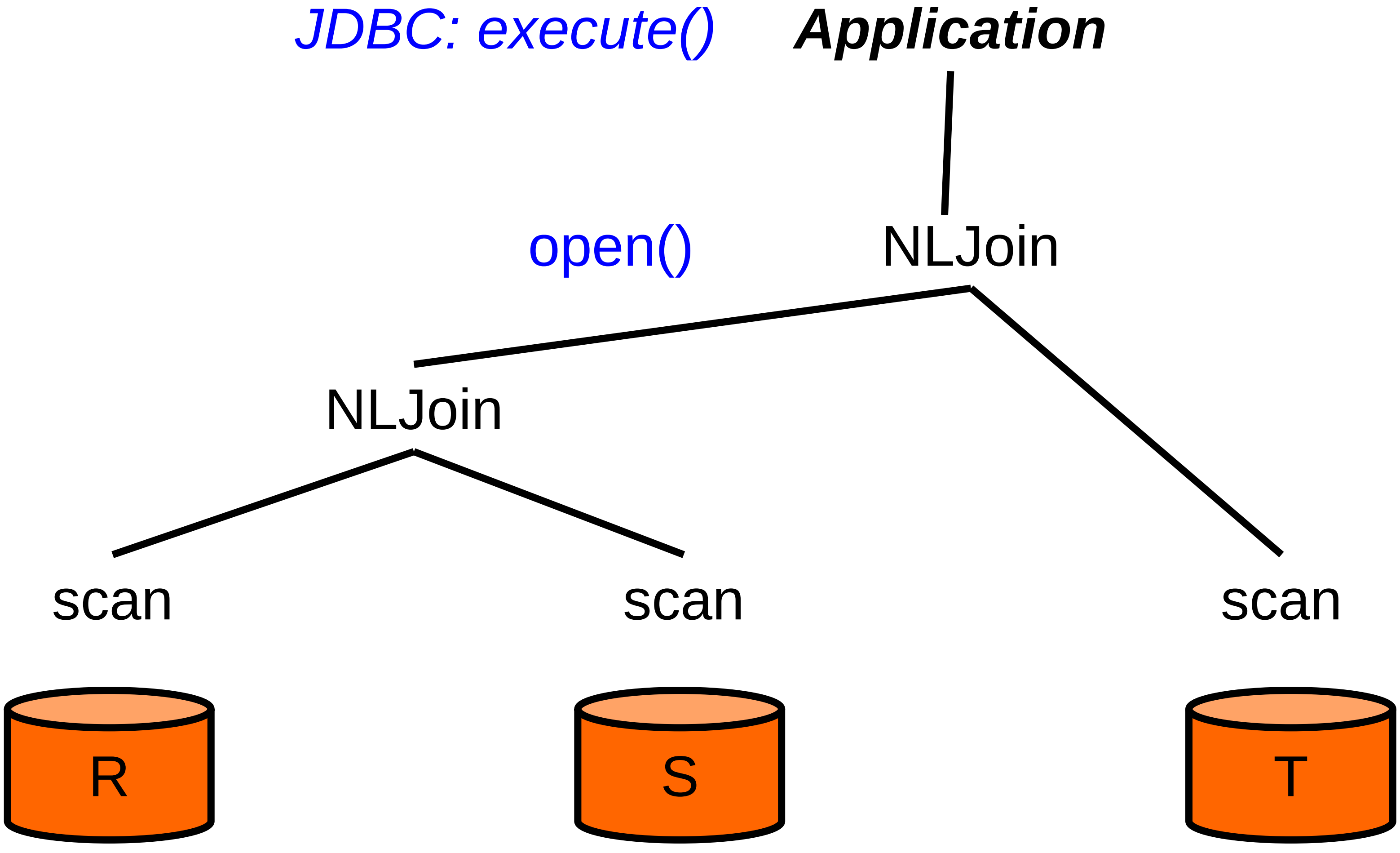
Iterator Model at Work



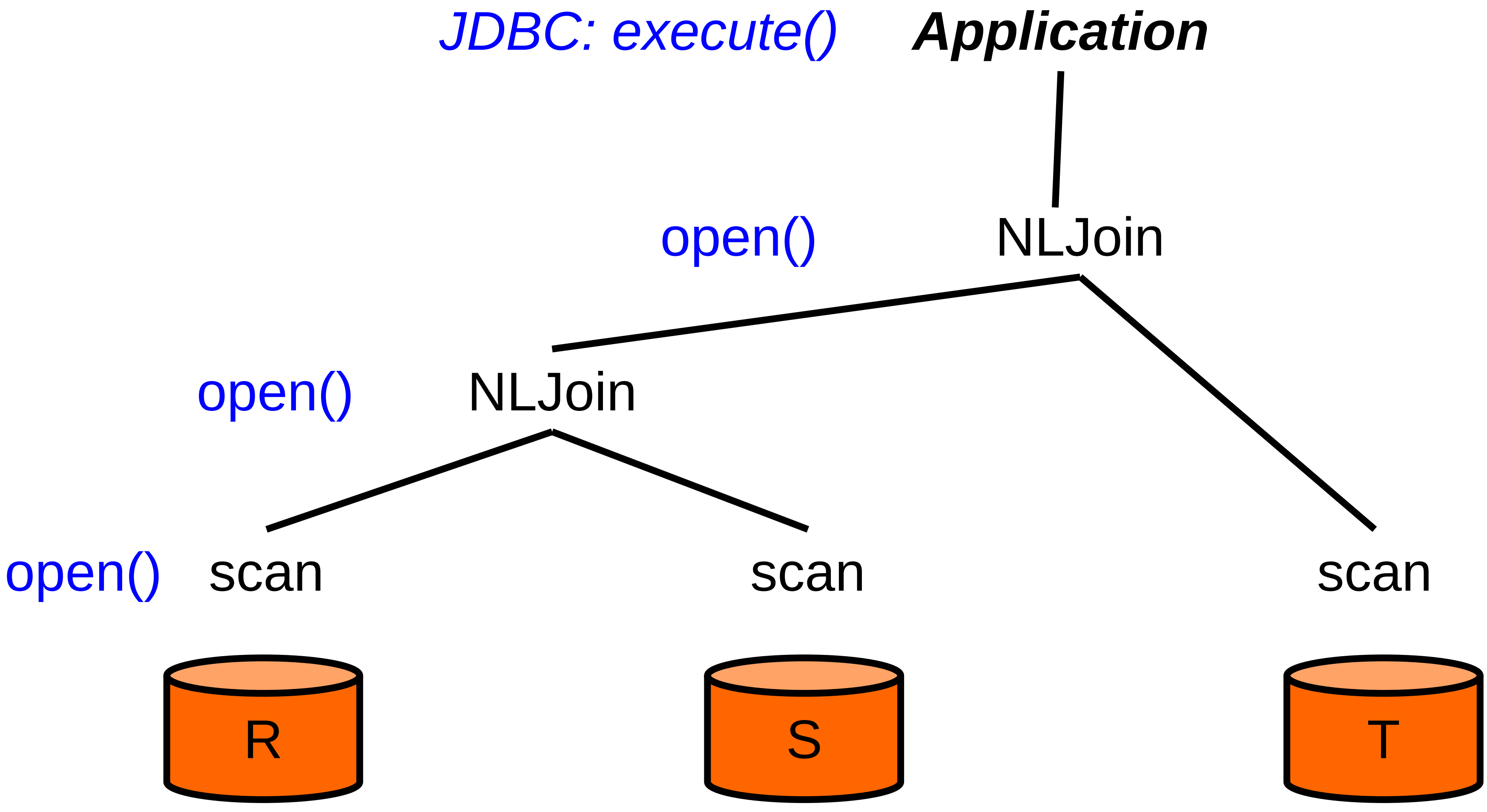
Iterator Model at Work



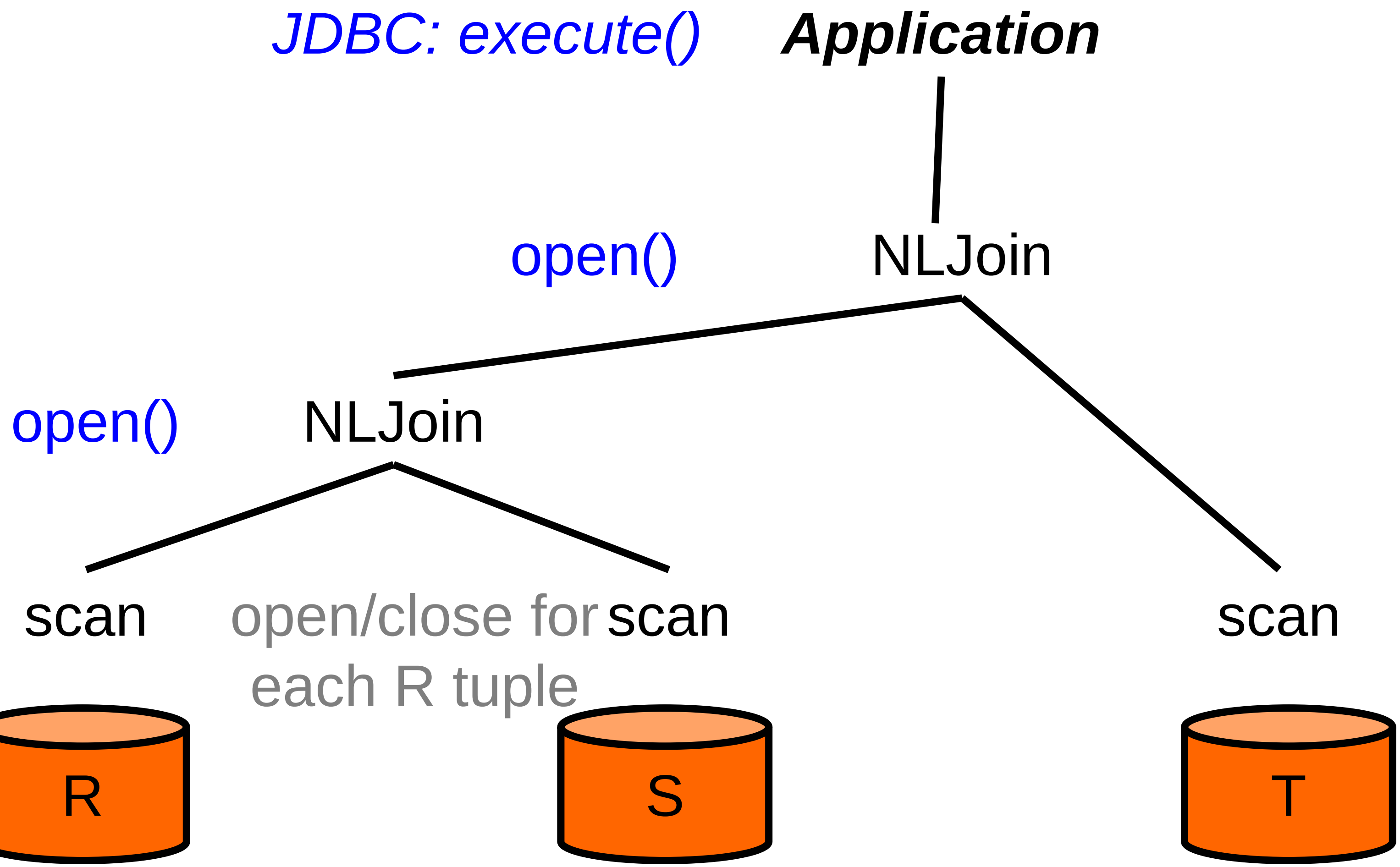
Iterator Model at Work



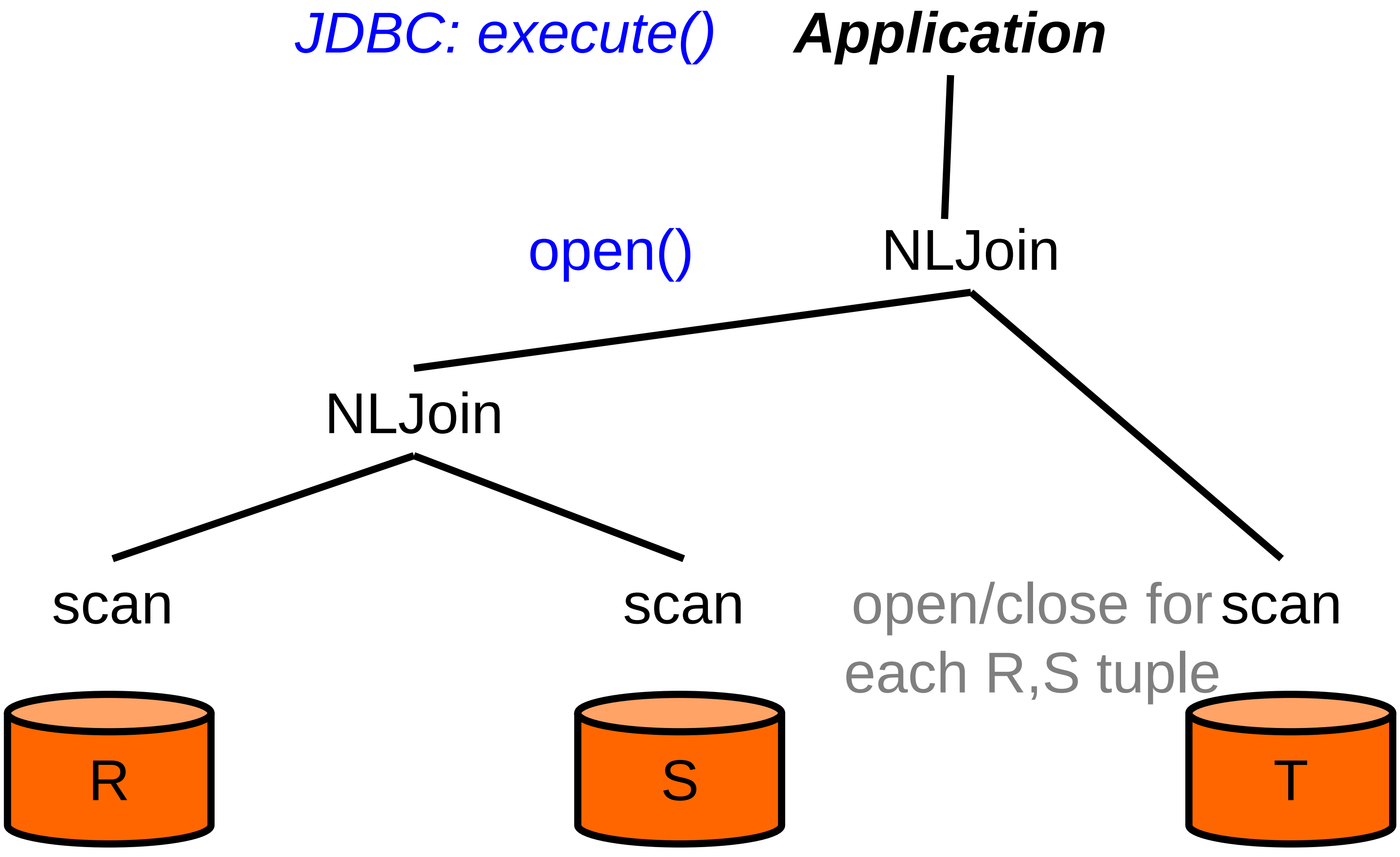
Iterator Model at Work



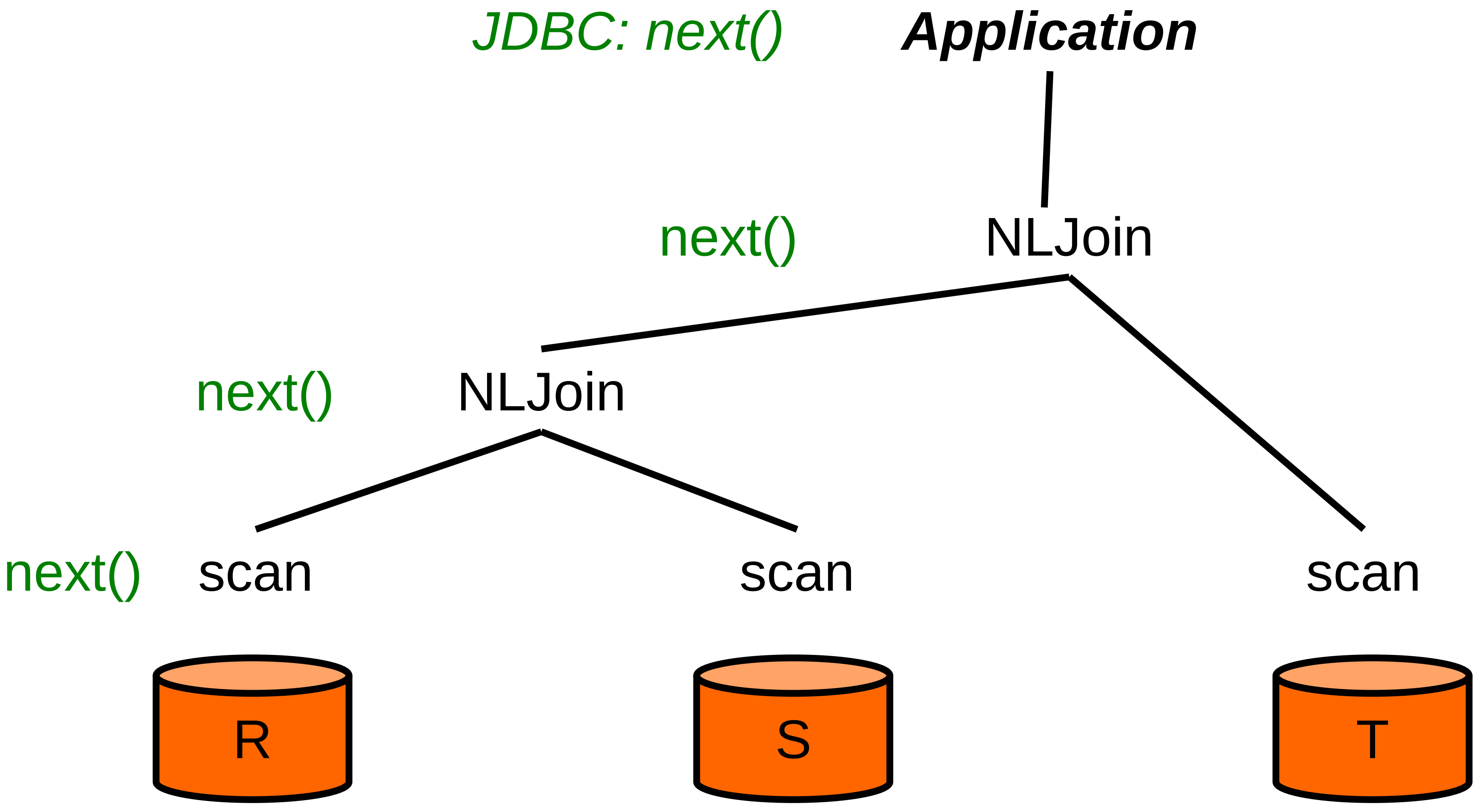
Iterator Model at Work



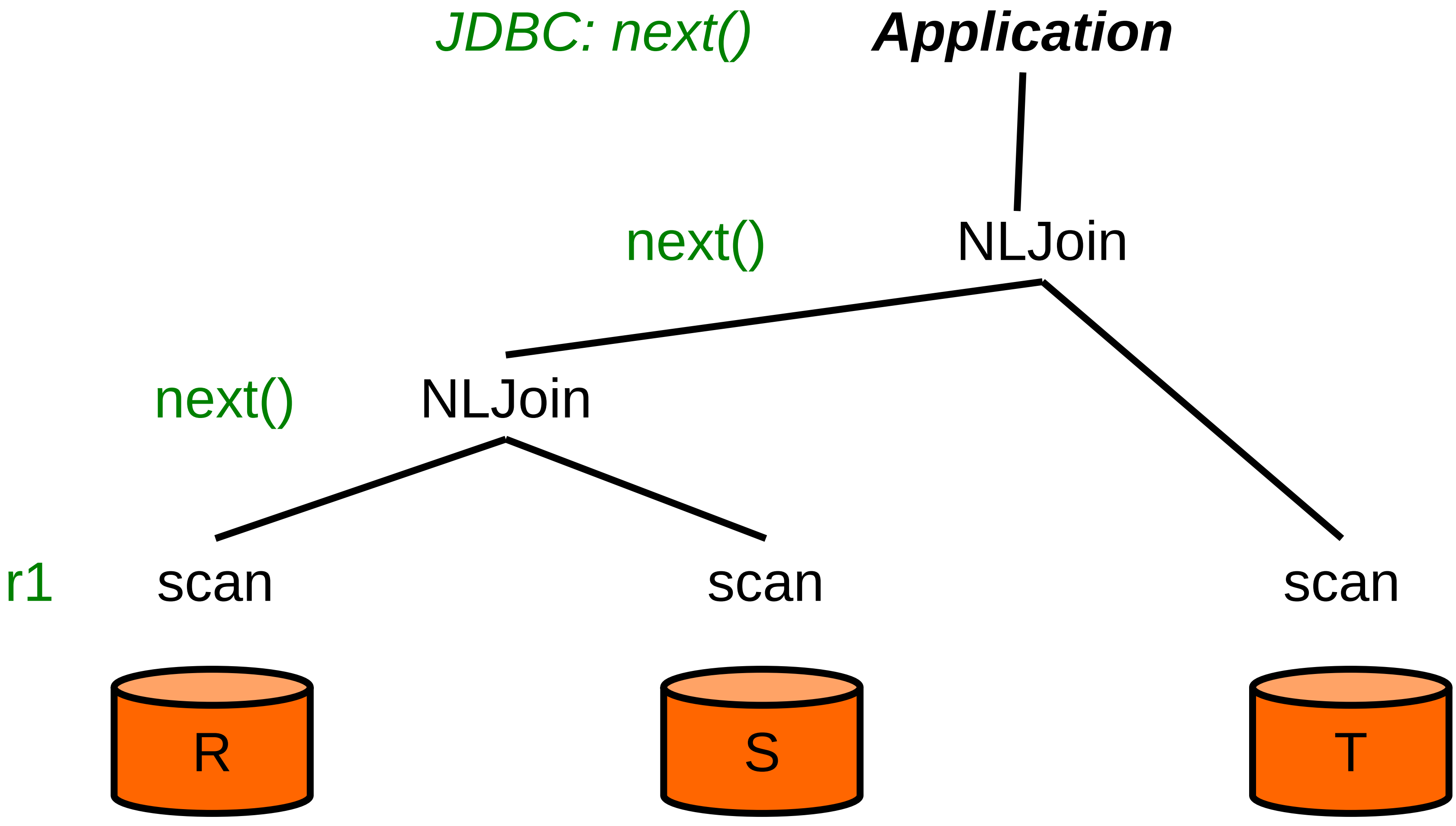
Iterator Model at Work



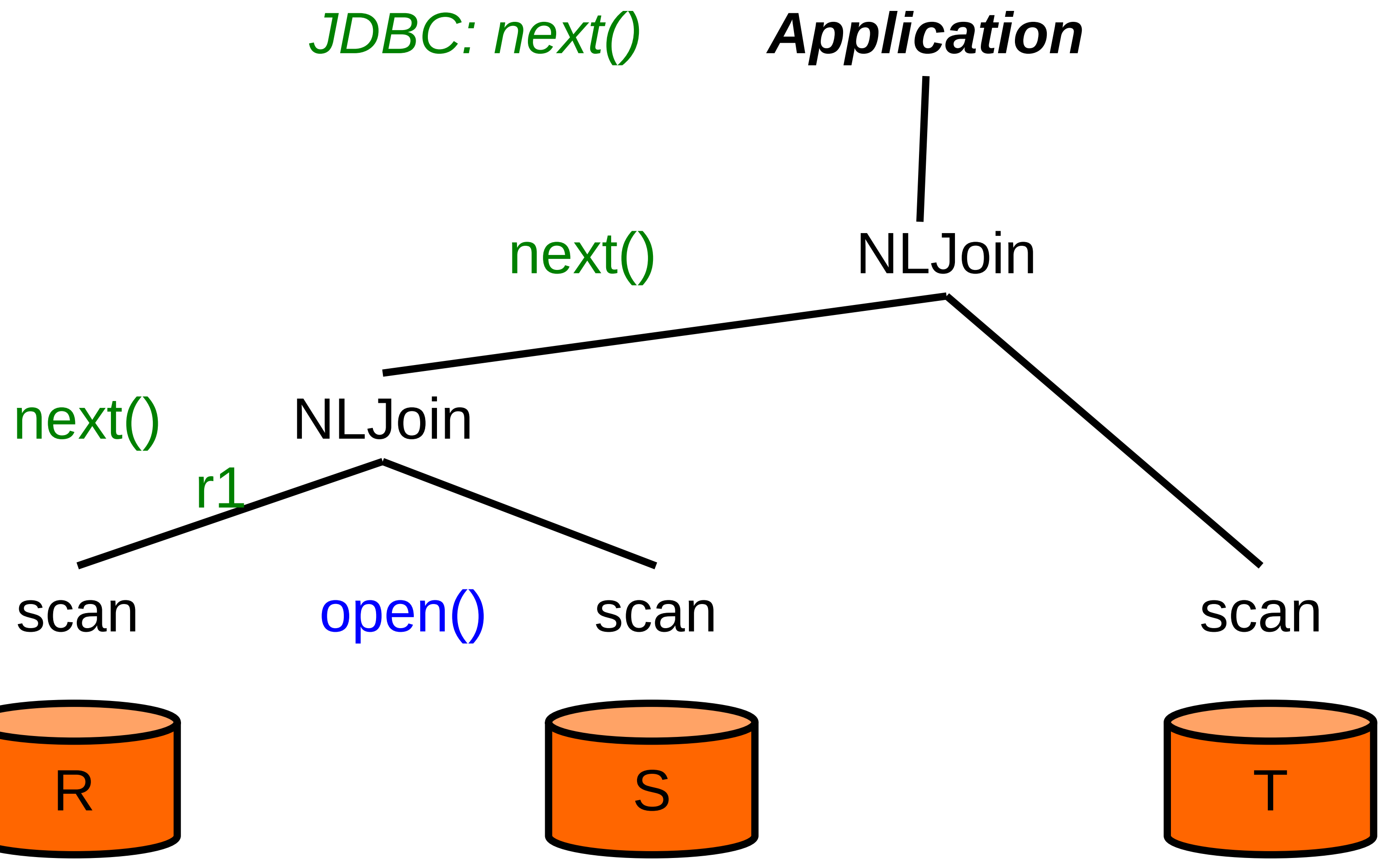
Iterator Model at Work



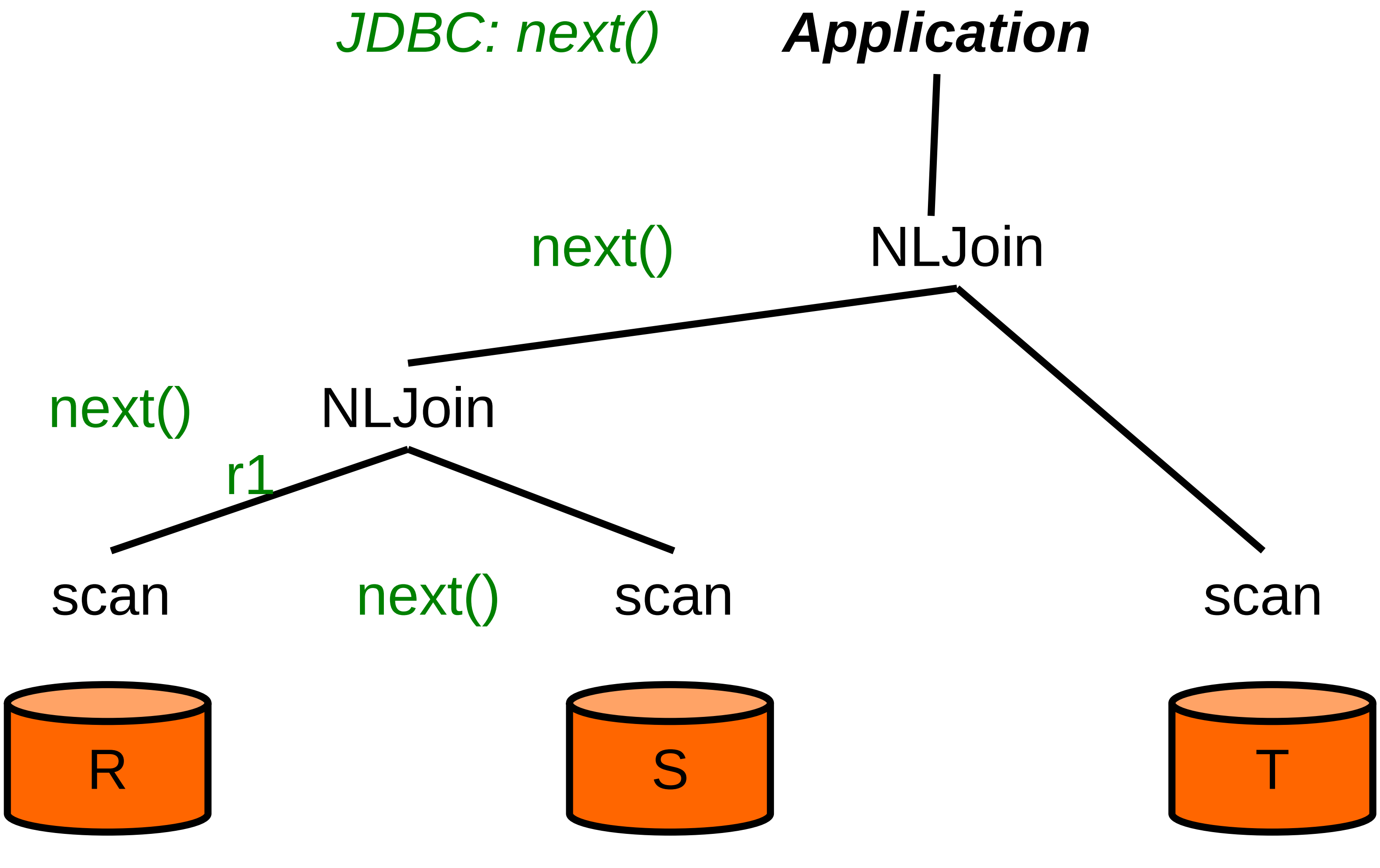
Iterator Model at Work



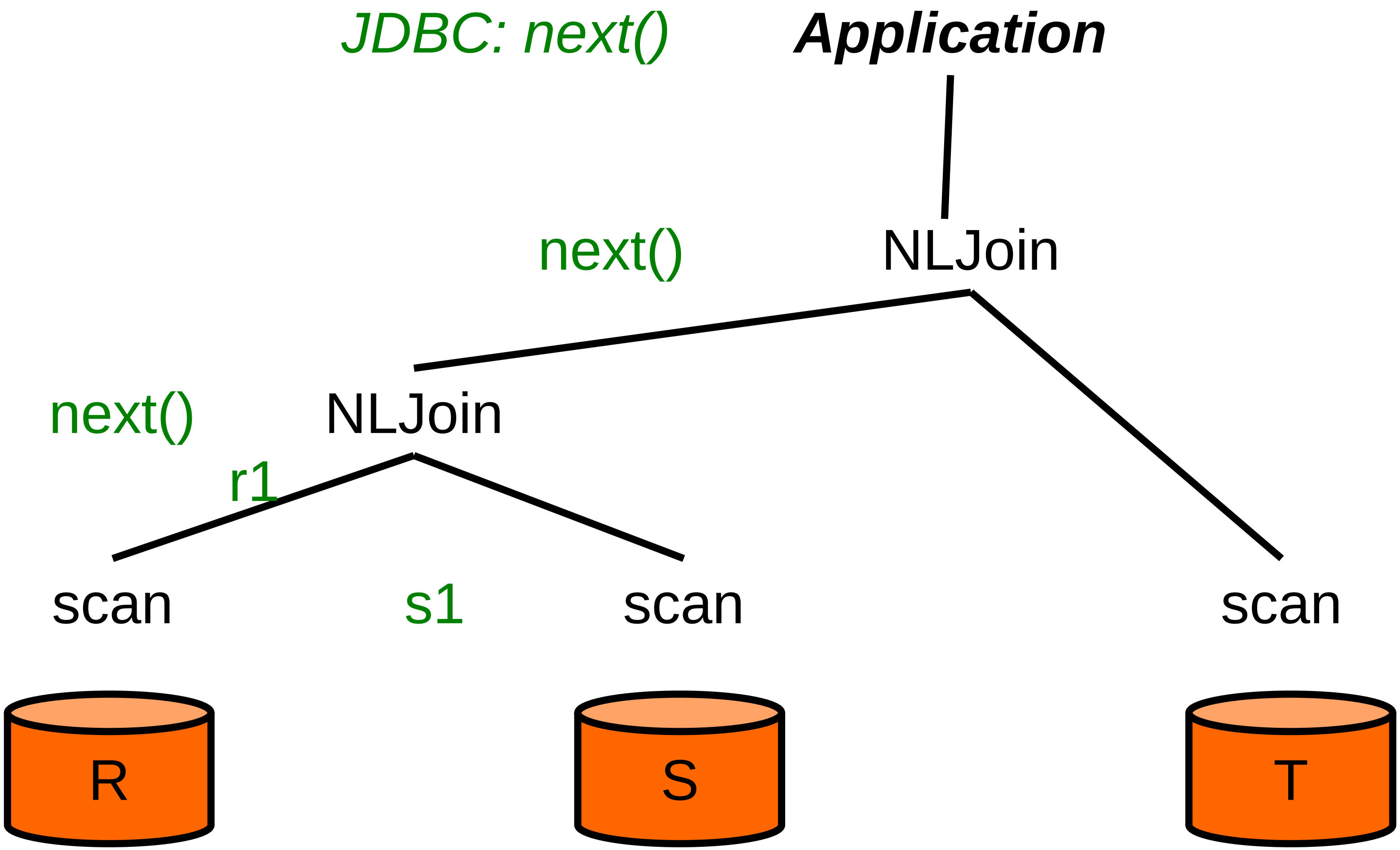
Iterator Model at Work



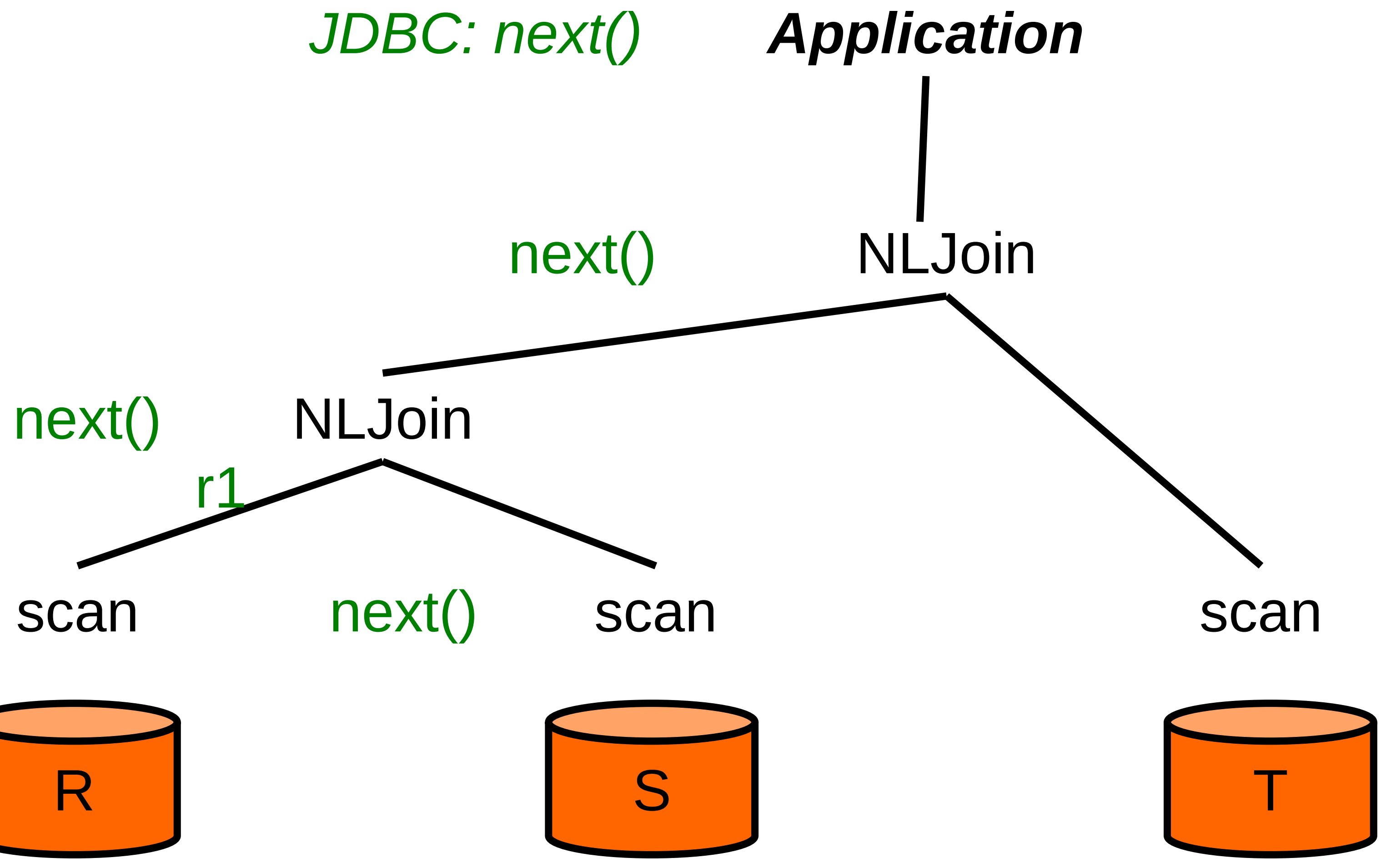
Iterator Model at Work



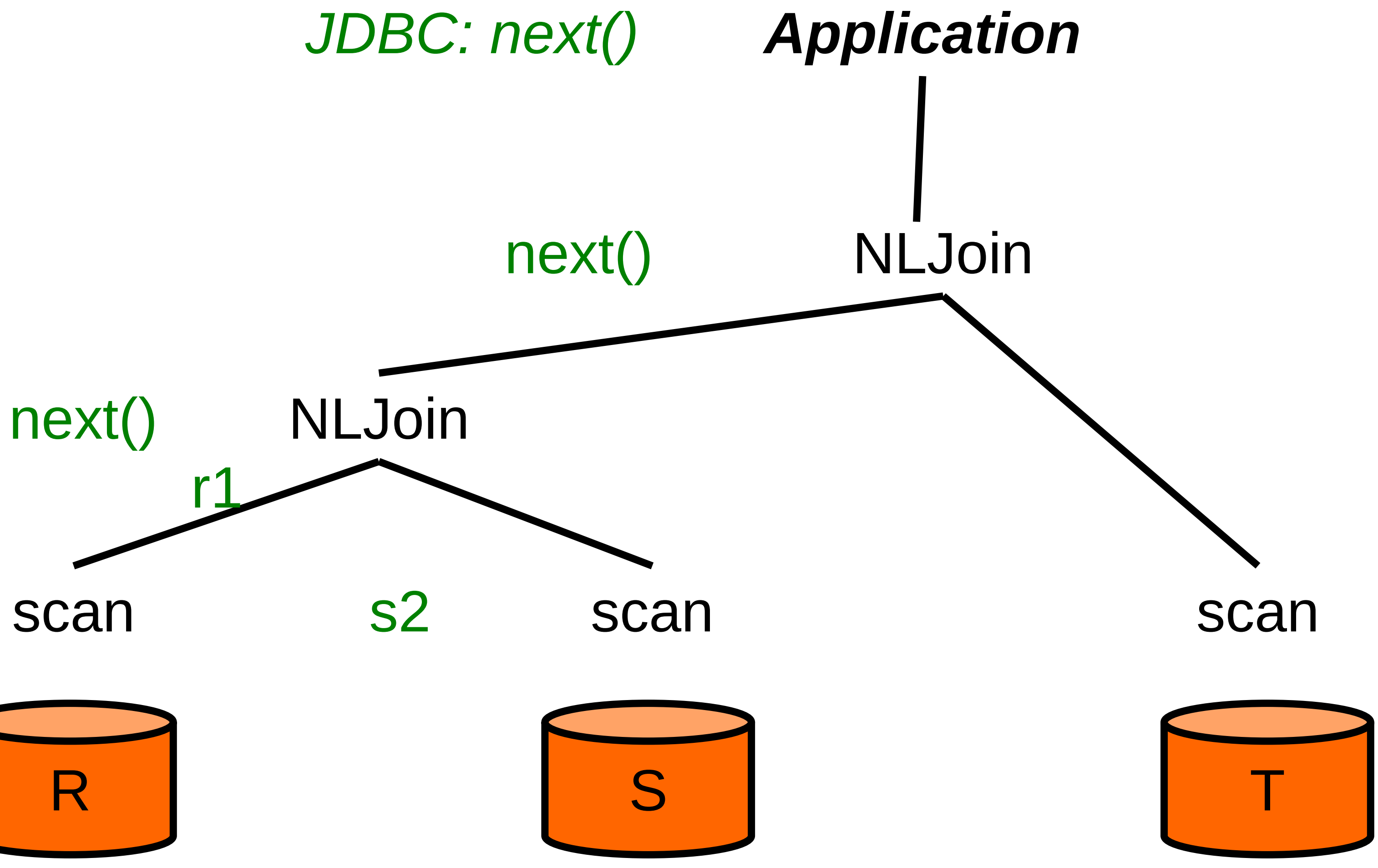
Iterator Model at Work



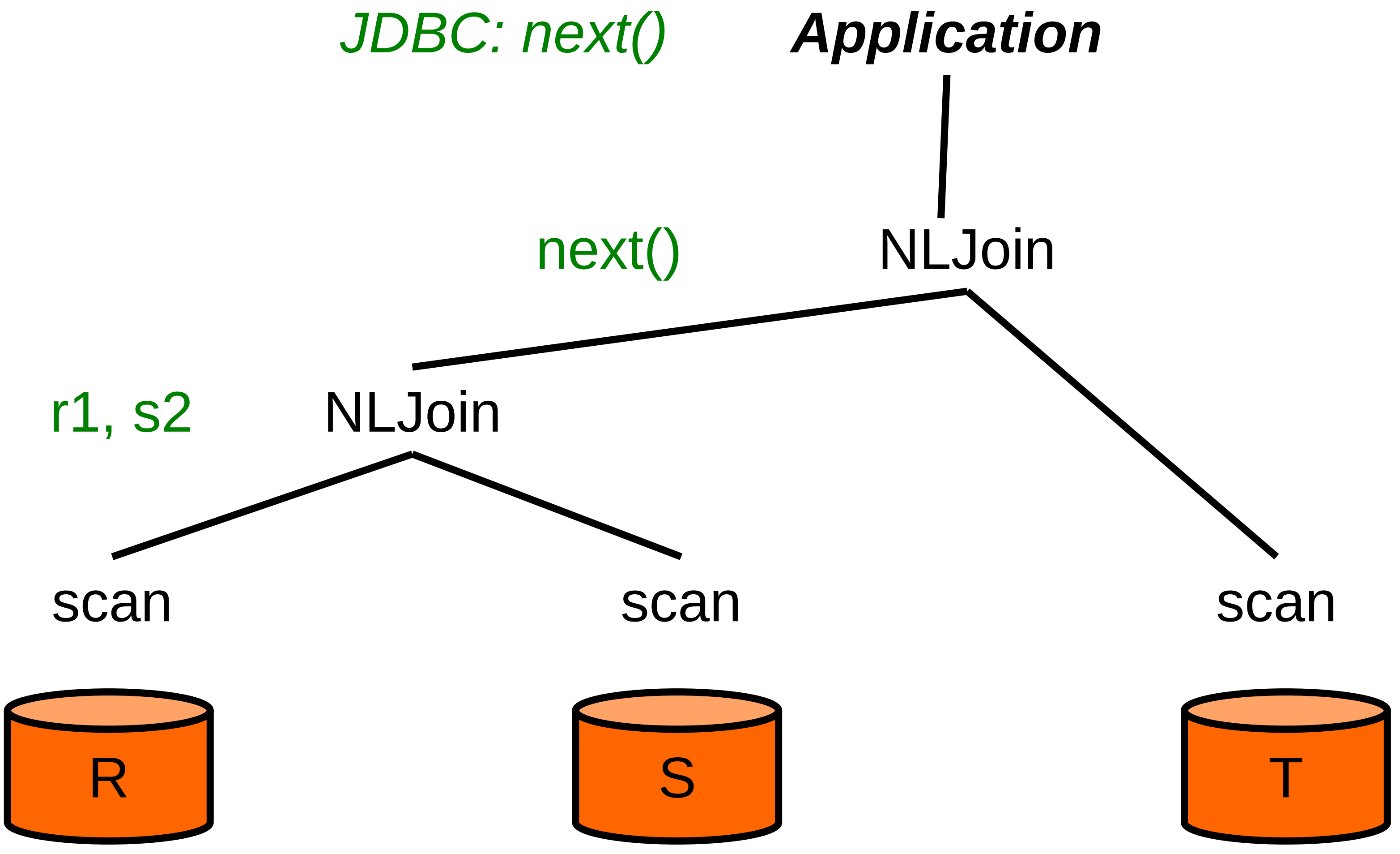
Iterator Model at Work



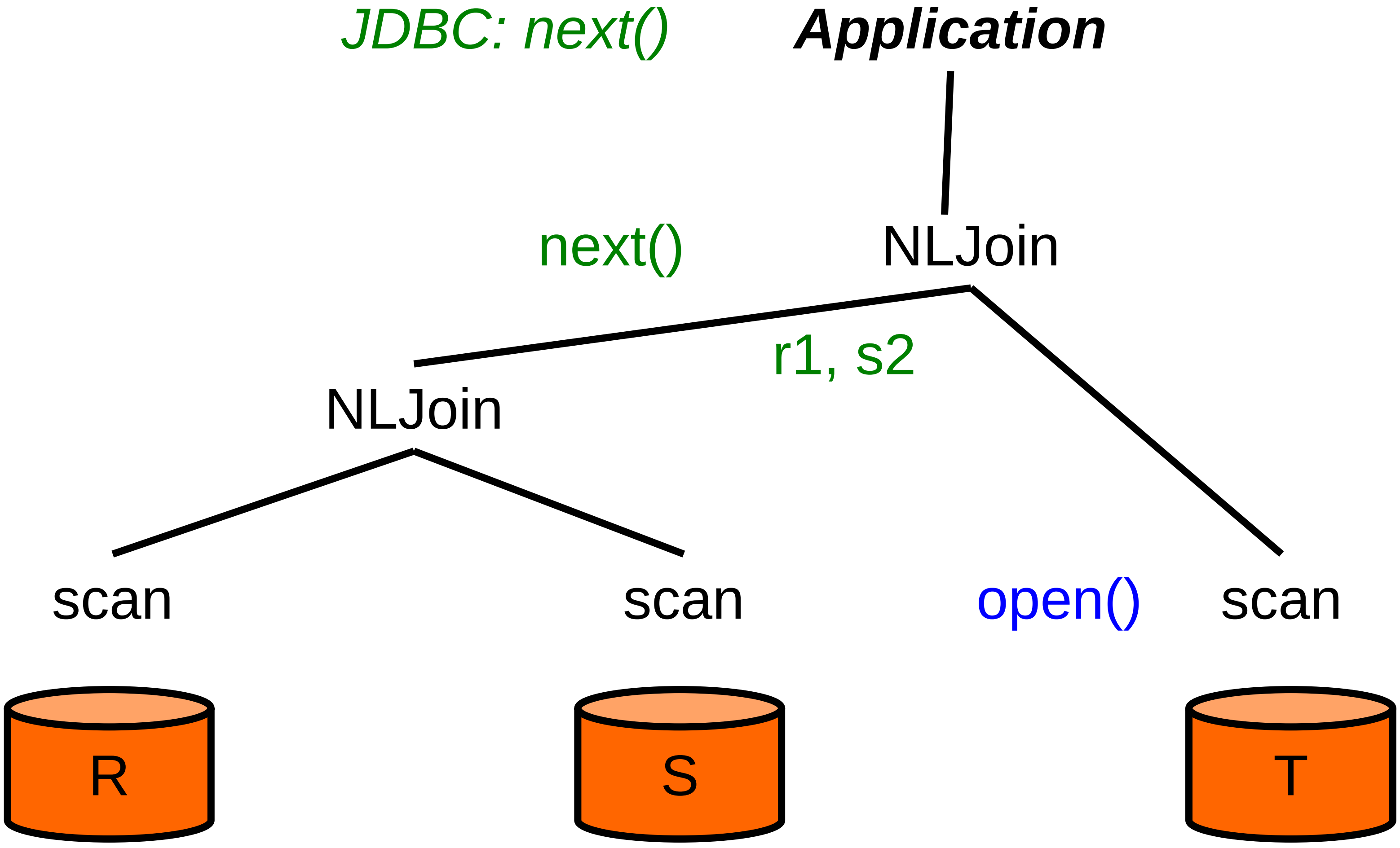
Iterator Model at Work



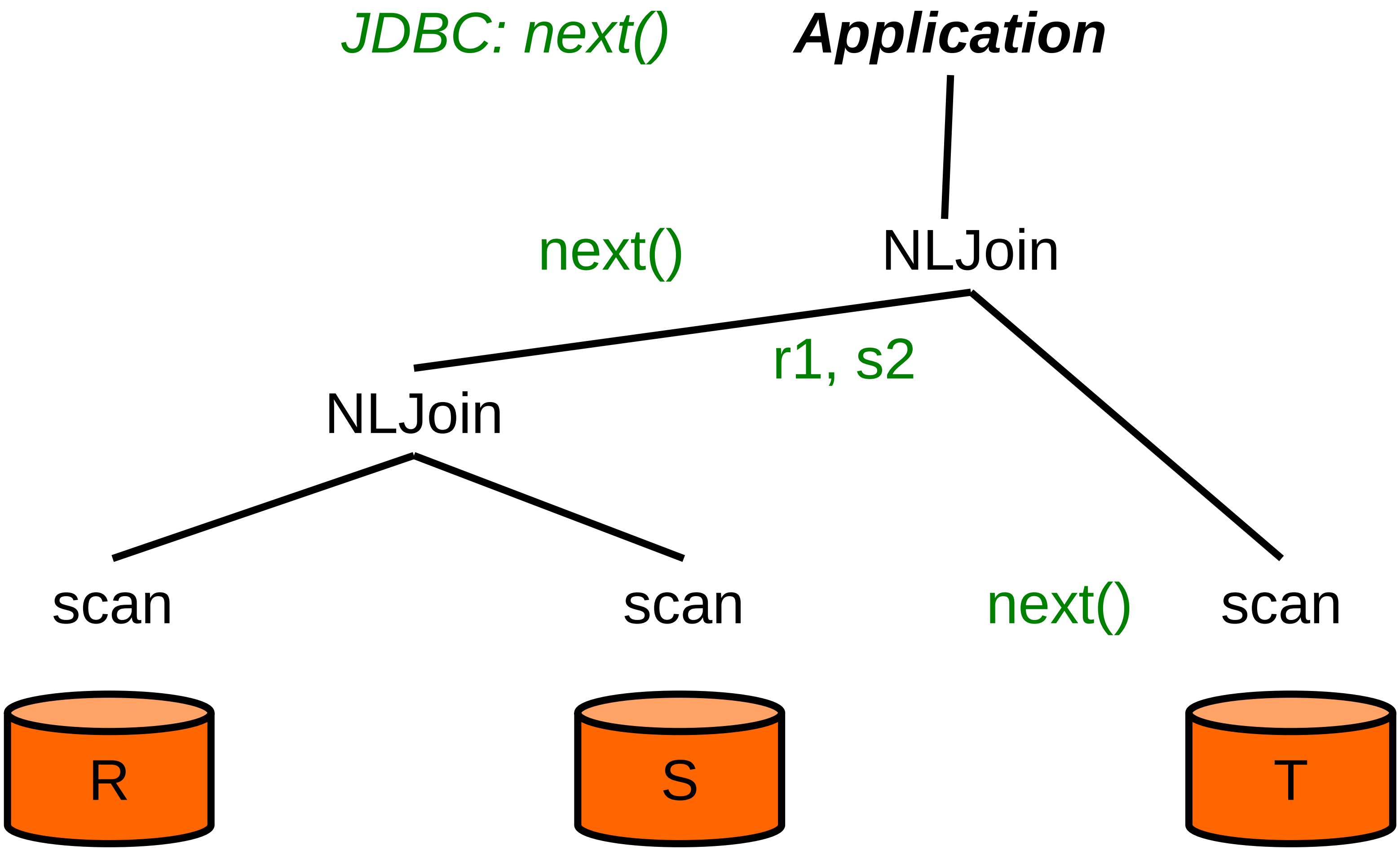
Iterator Model at Work



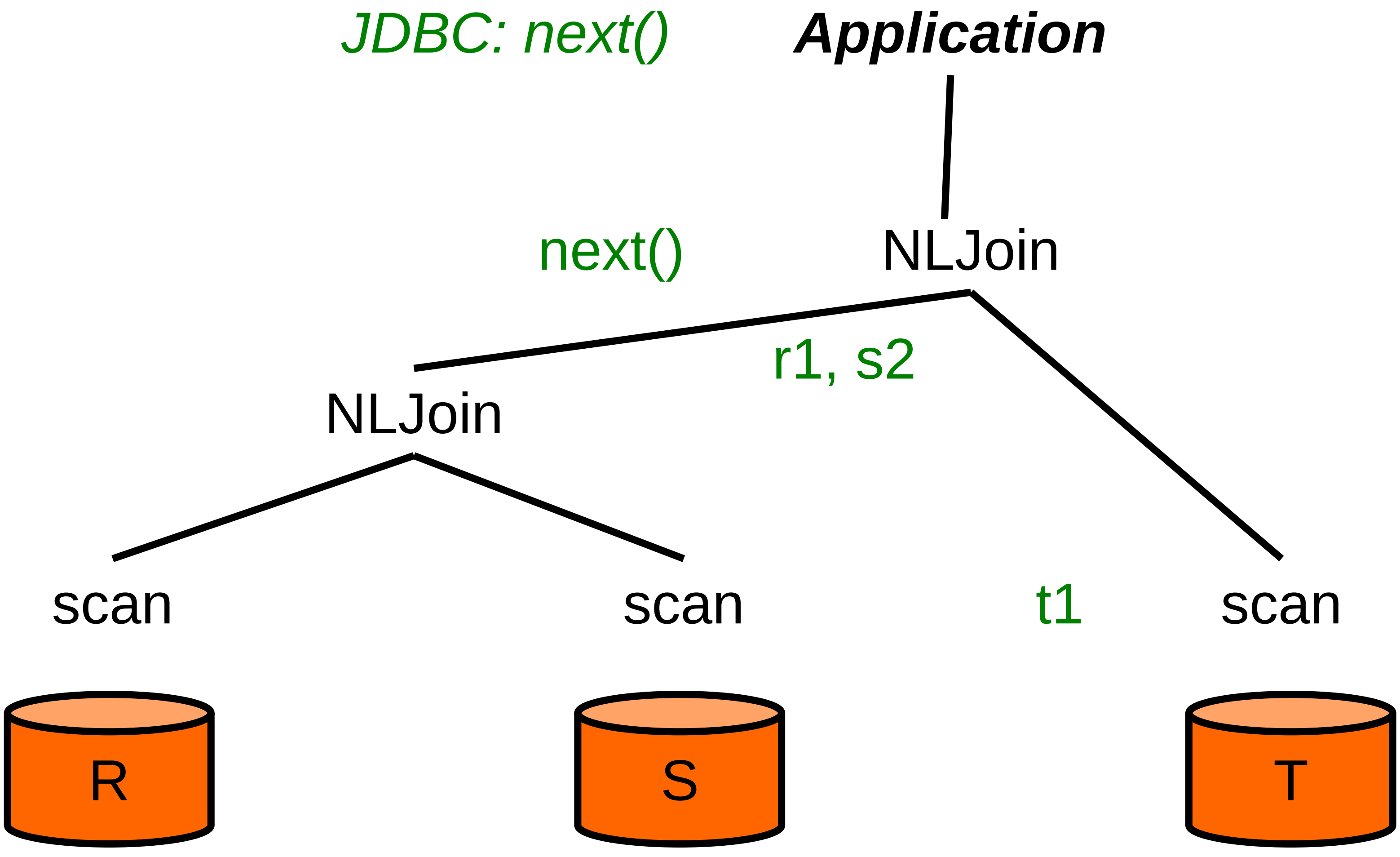
Iterator Model at Work



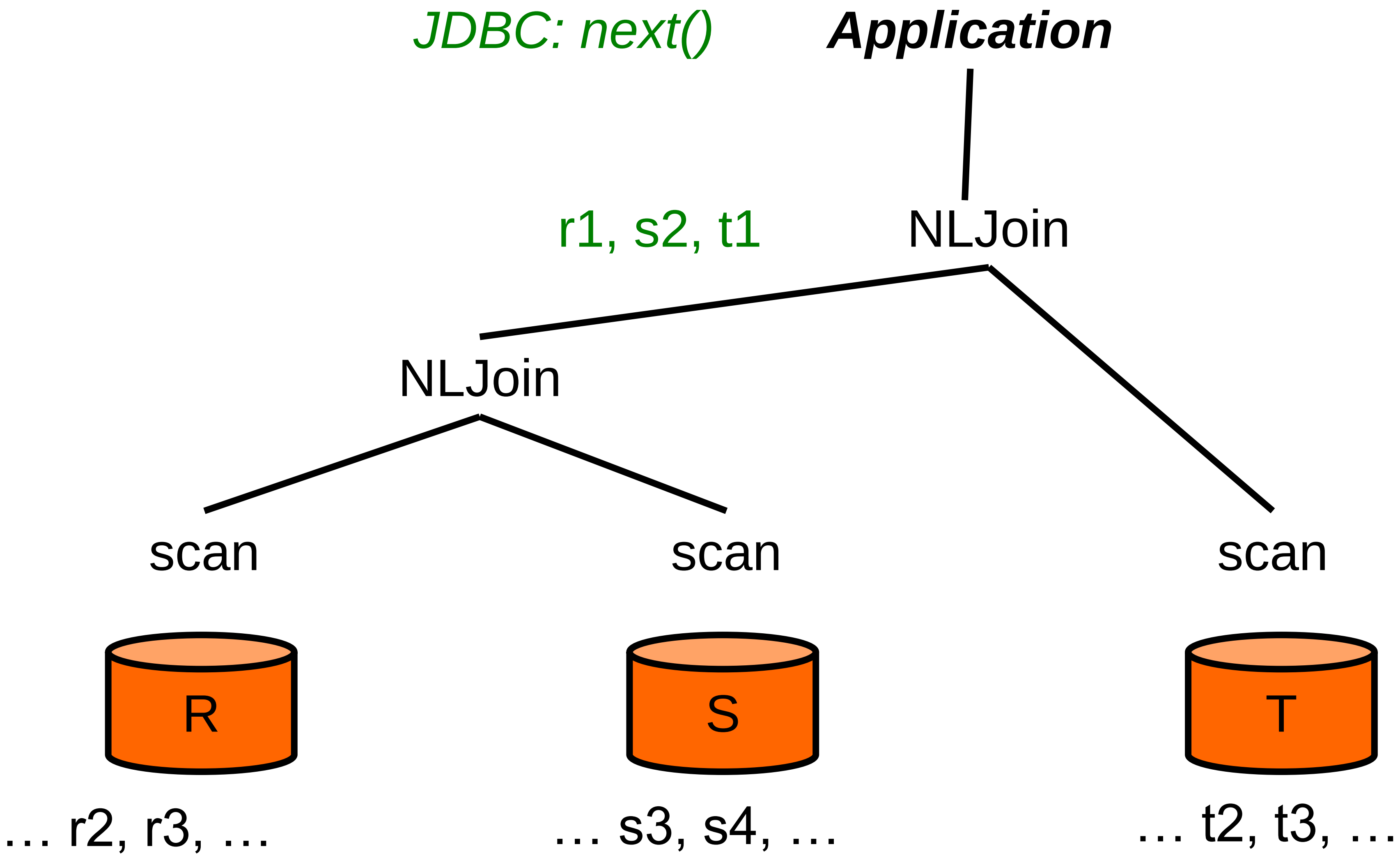
Iterator Model at Work



Iterator Model at Work



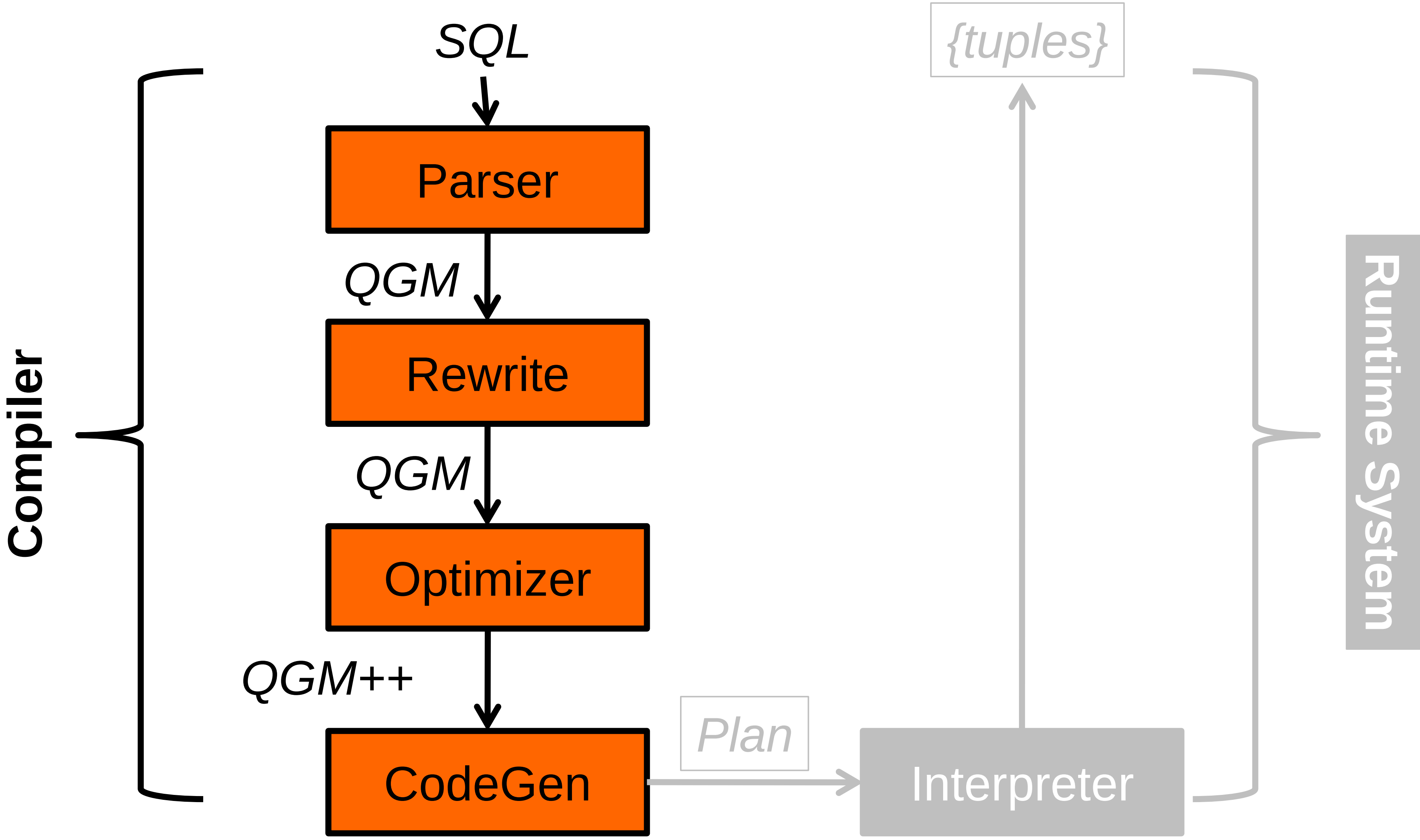
Iterator Model at Work



Iterators Summary: Easy & Costly

- Principle
  - data flows bottom up in a plan (i.e. operator tree)
  - control flows top down in a plan
- Advantages
  - generic interface for all operators: great information hiding
  - easy to implement iterators (clear what to do in any phase)
  - works well with JDBC and embedded SQL
  - supports DBmin and other buffer management strategies
  - no overheads in terms of main memory
  - supports pipelining: great if only subset of results consumed
  - supports parallelism and distribution: add special iterators
- Disadvantages
  - high overhead of method calls
  - poor instruction cache locality

Query Processor



SQL -> Relational Algebra

SQL

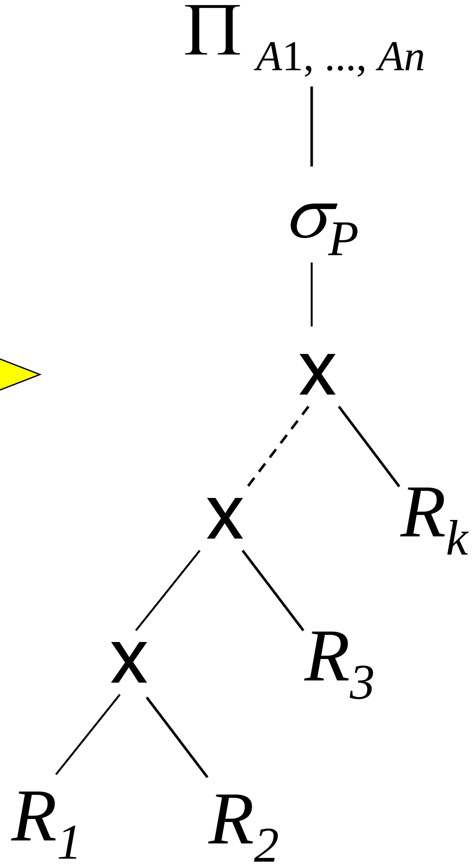
select $A_1, \ldots, A_n$

from $R_1, \ldots, R_k$

where P;

Relational Algebra

$$\Pi_{A1, \ldots, An}(\sigma_P (R_1 \times \ldots \times R_k))$$



SQL -> QGM

SQL

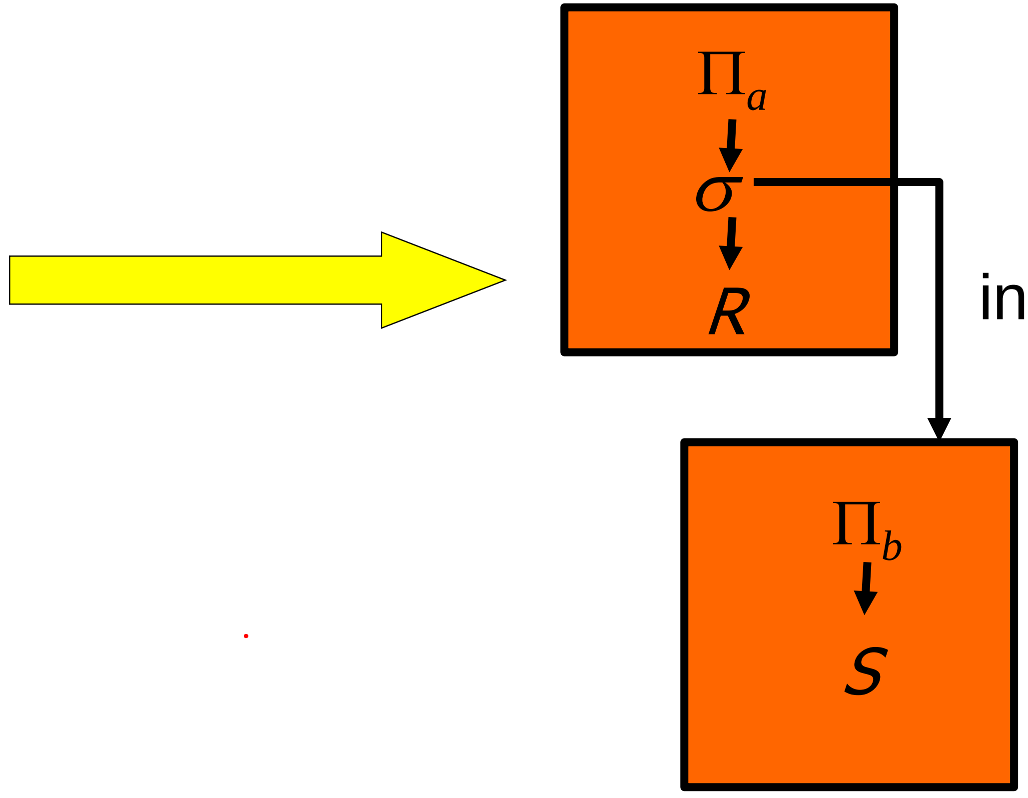
QGM

select *a*

from *R*

where *a in (***select** *b*

from *S);*



Parser

- Generates rel. alg. tree for each sub-query
  - constructs graph of trees: Query Graph Model (QGM)
  - nodes are subqueries
  - edges represent relationships between subqueries
- Extended rel. algebra because SQL more than RA
  - GROUP BY: Γ operator
  - ORDER BY: sort operator
  - DISTINCT: can be implemented with Γ operator
- Parser needs schema information
  - Why? Give examples.

- Why can`t a query be compiled into one tree?

SQL -> Relational Algebra

SQL

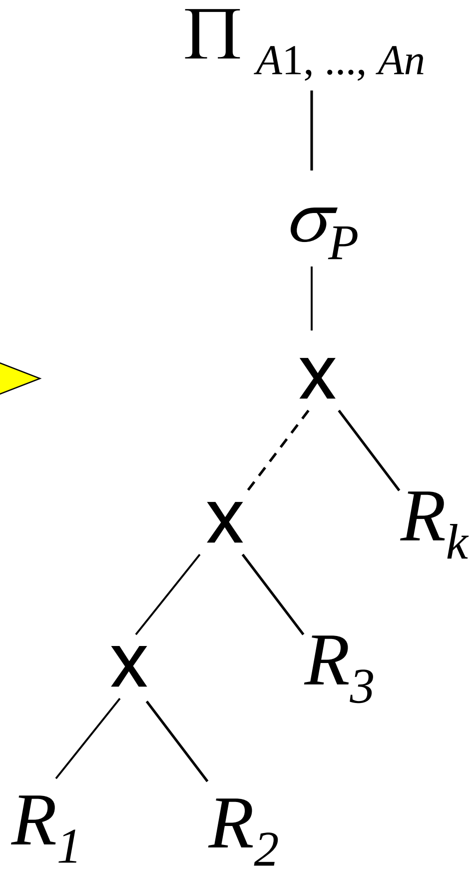
Relational Algebra

$$\Pi_{A1, \ldots, An}(\sigma_P (R_1 \times \ldots \times R_k))$$

select $A_1, \ldots, A_n$

from $R_1, \ldots, R_k$

where P;



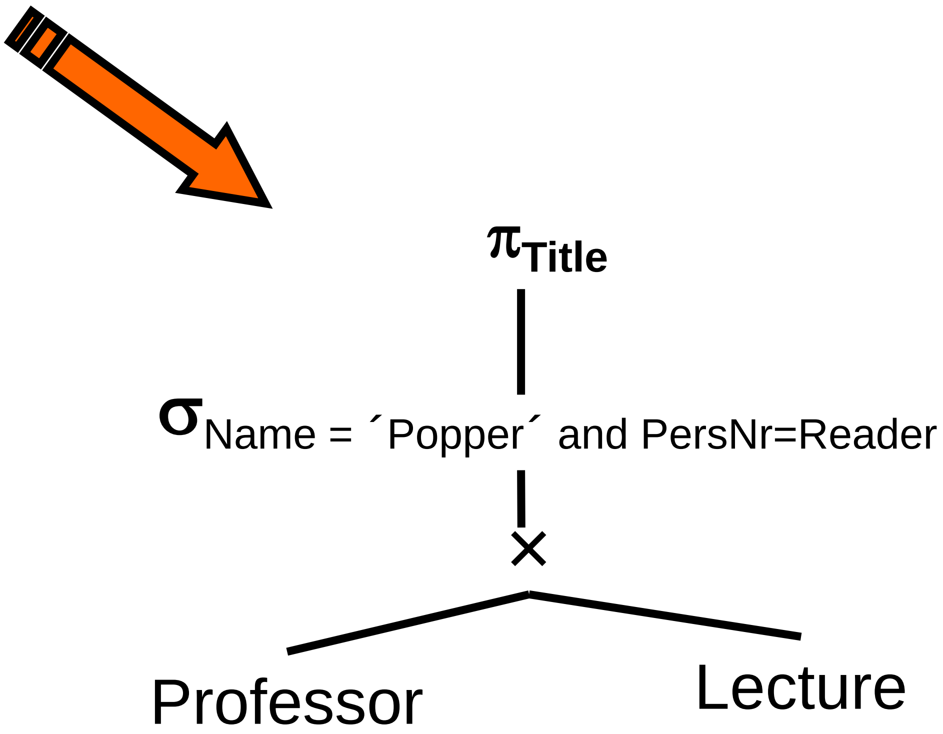
Example: SQL -> Relational Algebra

select Title

from Professor, Lecture

where Name = ´Popper´ **and**

PersNr = Reader



$\pi_{\text{Title}} (\sigma_{\text{Name = ´Popper´ and PersNr=Reader}} (\text{Professor} \times \text{Lecture}))$

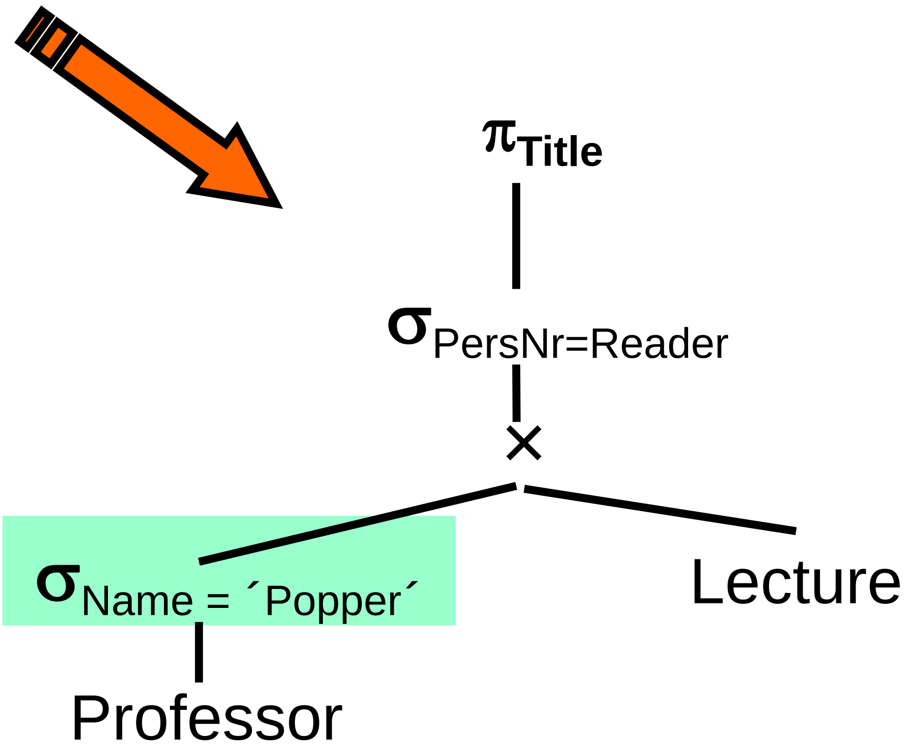
First Optimization: Push-down σ

select Title

from Professor, Lecture

where Name = ´Popper´ **and**

PersNr = Reader



$\pi_{\text{Title}}\ (\sigma_{\text{PersNr=Reader}}\ ((\sigma_{\text{Name = ´Popper´}}\ \text{Professor}) \times \text{Lecture}))$

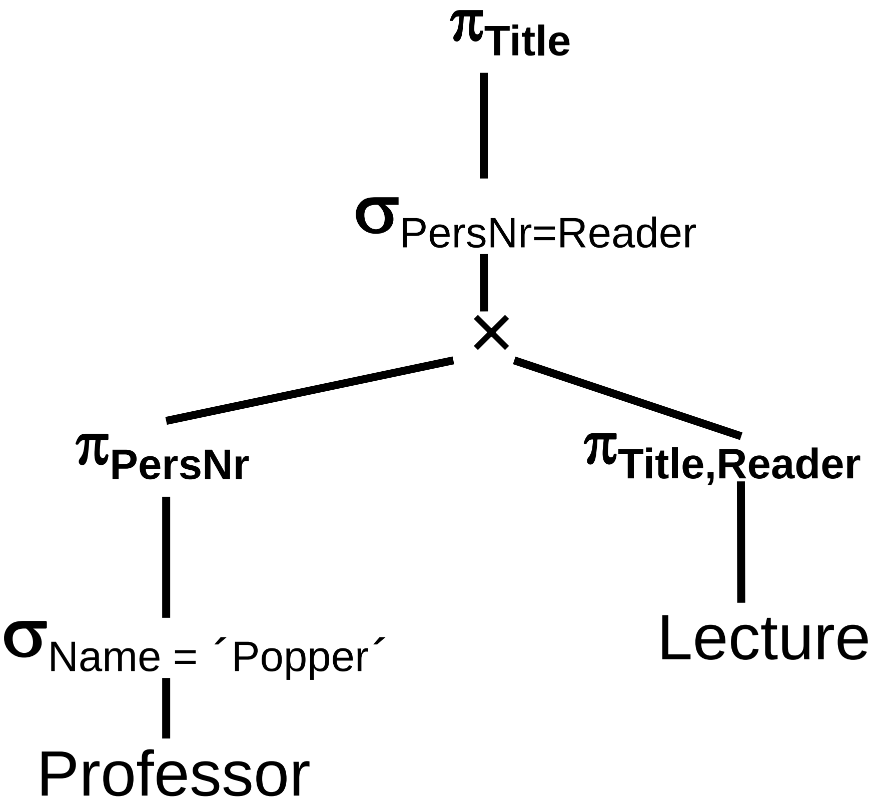
Second Optimization: Push-down π

select Title

from Professor, Lecture

where Name = ´Popper´ **and**

PersNr = Reader



Correctness: Push-down π

- $\pi_{\text{Title}} (\sigma_{\text{PersNr=Reader}} ((\sigma_{\text{Name = ´Popper´}} \text{Professor}) \times \text{Lecture}))$

  (composition of projections)

- $\pi_{\text{Title}} (\pi_{\text{Title,PersNr,Reader}} (\sigma_{\ldots} ((\sigma_{\ldots} \text{Professor}) \times \text{Lecture})))$

  (commutativity of π and σ)

- $\pi_{\text{Title}} (\sigma_{\ldots} (\pi_{\text{Title,PersNr,Reader}} ((\sigma_{\ldots} \text{Professor}) \times \text{Lecture})))$

  (commutativity of π and σ)

- $\pi_{\text{Title}} (\sigma_{\ldots} (\pi_{\text{PersNr}} (\sigma_{\ldots} \text{Professor}) \times \pi_{\text{Title,Reader}} (\text{Lecture})))$

Second Optimization: Push down π

- Correctness (see previous slide – example generalizes)

- Why is it good? (almost same reason as for σ
  - reduces size of intermediate results
  - but: only makes sense if results are materialized; e.g. sort
    - does not make sense if pointers are passed around in iterators

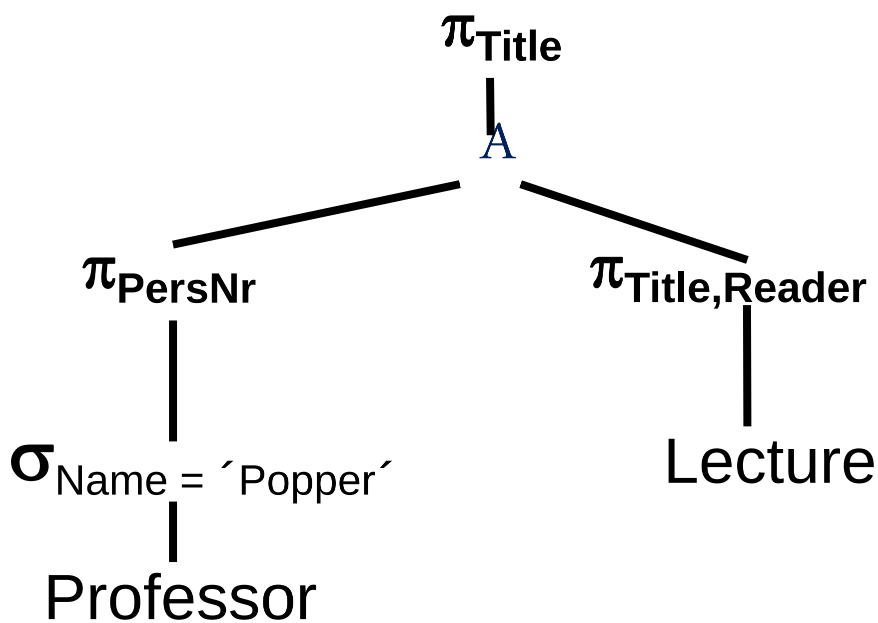
Third Optimization: σ + x = A

select Title

from Professor, Lecture

where Name = ´Popper´ **and**

PersNr = Reader



Third Optimization: σ + x = A

- Correctness by definition of A operator
- Why is this good?
  - x always done using nested-loops algorithm
    - A can also be carried out using hashing, sorting, index support
    - choice of better algorithm may result in huge wins
  - x produces large intermediate results
    - results in a huge number of „next()“ calls in iterator model
    - method calls are expensive
- Selection, projection push-down are no-brainers
  - make sense whenever applicable
  - do not need a cost model to decide how to apply them
  - (exception: expensive selections, projections with UDF)
  - done in a phase called query rewrite, based on rules
- More complex query rewrite rules…

Unnesting of Views

- Example: Unnesting of Views

```
select A.x
from   A
where y in
   (select y from B)
```

```
select A.x
from   A, B
where A.y = B.y
```

- Example: Unnesting of Views

```
select A.x
from   A
where exists
   (select * from B where A.y = B-y)
```

```
select A.x
from   A, B
where A.y = B.y
```

- Is this correct? Why is this better?
  - (not trivial at all!!!)

Query Rewrite

- Example: Predicate Augmentation

```
select *
from   A, B, C
where  A.x = B.x
       and B.x = C.x
```

```
select *
from   A, B, C
where  A.x = B.x
       and B.x = C.x
       and A.x = C.x
```

Why is that useful?

Pred. Augmentation: Why useful?

A (odd numbers)

...	x
...	1
...	3
...	5
...	...

B (all numbers)

...	x
...	1
...	2
...	3
...	...

C (even numbers)

...	x
...	2
...	4
...	6
...	...

- Cost((A A C) A B) < Cost((A A B) A C)
  - get second join for free
- Query Rewrite does not know that, …
  - but it knows that it might happen and hopes for optimizer…
- Codegen gets rid of unnecessary predicates (e.g., A.x = B.x)

Query Optimization

- Two tasks
  - Determine order of operators
  - Determine algorithm for each operator (hashing, sorting, …)
- Components of a query optimizer
  - Search space
  - Cost model
  - Enumeration algorithm
- Working principle
  - Enumerate alternative plans
  - Apply cost model to alternative plans
  - Select plan with lowest expected cost

Query Optimization: Does it matter?

- A x B x C
  - size(A) = 10,000
  - size(B) = 100
  - size(C) = 1
  - cost(X x Y) = size(X) + size(Y)

- cost((A x B) x C) = **1,010,001**
  - cost(A x B) = 10,100
  - cost(X x C) = 1,000,001 with X = A x B

- cost (A x (B x C)) = **10,201**
  - cost(B x C) = 101
  - cost(A x X) = 10,100 with X = B x C

Query Opt.: Does it matter?

- A x B x C
  - size(A) = 1000
  - size(B) = 1
  - size(C) = 1
  - **cost(X x Y) = size(X) * size(Y)**
- cost((A x B) x C) = **2000**
  - cost(A x B) = 1000
  - cost(X x C) = 1000 with X = A x B
- cost (A x (B x C)) = **1001**
  - cost(B x C) = 1
  - cost(A x X) = 1000 with X = B x C

Search Space: Relational Algebra

- Associativity of joins:
  (A A B) A C = A A (B A C)

- Commutativity of joins:
  A A B = B A A

- Many more rules
  - see Kemper/Eickler or Garcia-Molina text books

- What is better: A A B or B A A?
  - it depends
  - need cost model to make decision

Search Space: Group Bys

SELECT … FROM R, S WHERE R.a = S.a GROUP BY R.a, S.b;

- $\Gamma_{R.a,\ S.b}(R \text{ A } S)$

- $\Gamma_{S.b}(\Gamma_{R.a}(R) \text{ A } S)$

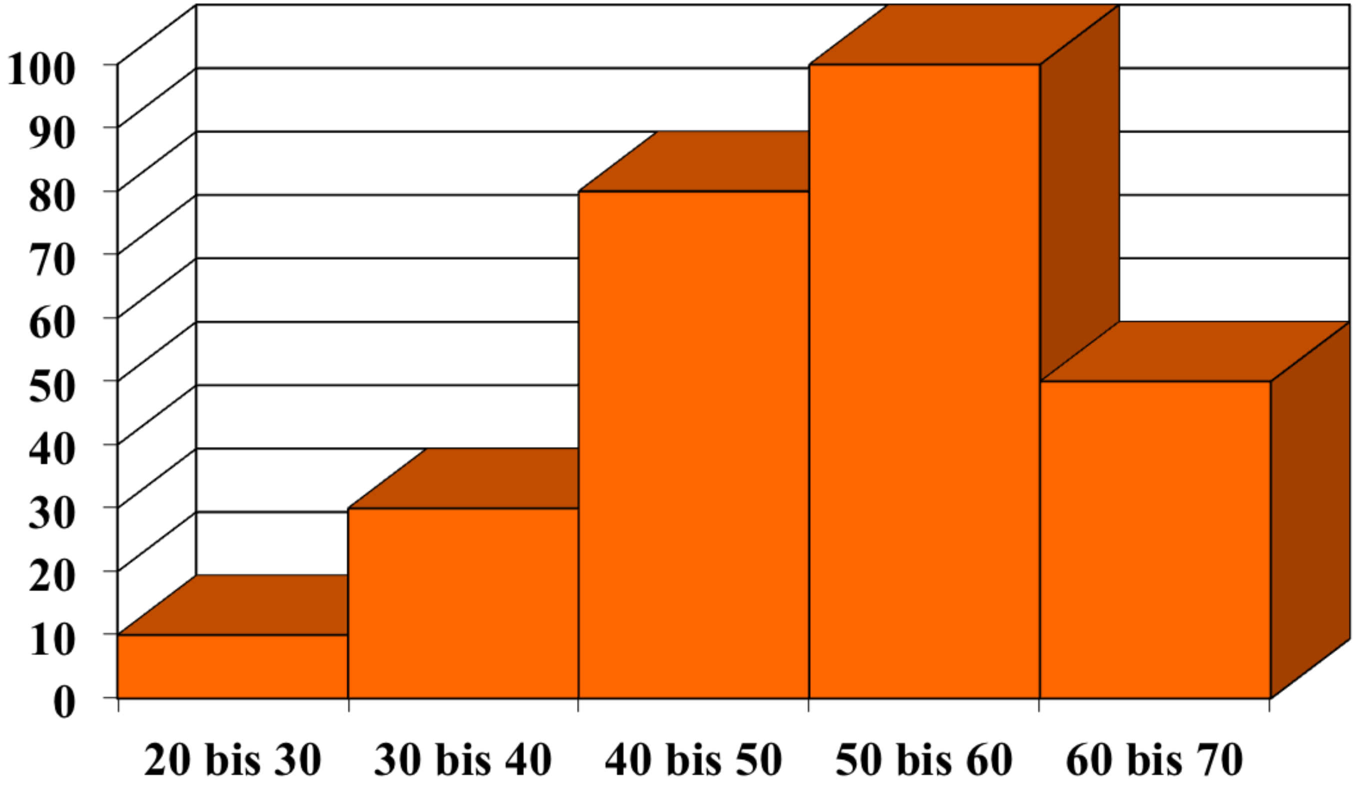
- Often, many possible ways to split & move group-bys
  - again, need cost model to make right decisions

Cost Model

- Cost Metrics
  - Response Time (consider parallelism)
  - Resource Consumption: CPU, IO, network
  - $ (often equivalent to resource consumption)
- Principle
  - Understand algorithm used by each operator (sort, hash, …)
    - estimate available main memory buffers
    - estimate the size of inputs, intermediate results
  - Combine cost of operators:
    - sum for resource consumption
    - max for response time (but keep track of bottlenecks)
- Uncertainties
  - estimates of buffers, interference with other operators
  - estimates of intermediate result size (histograms)

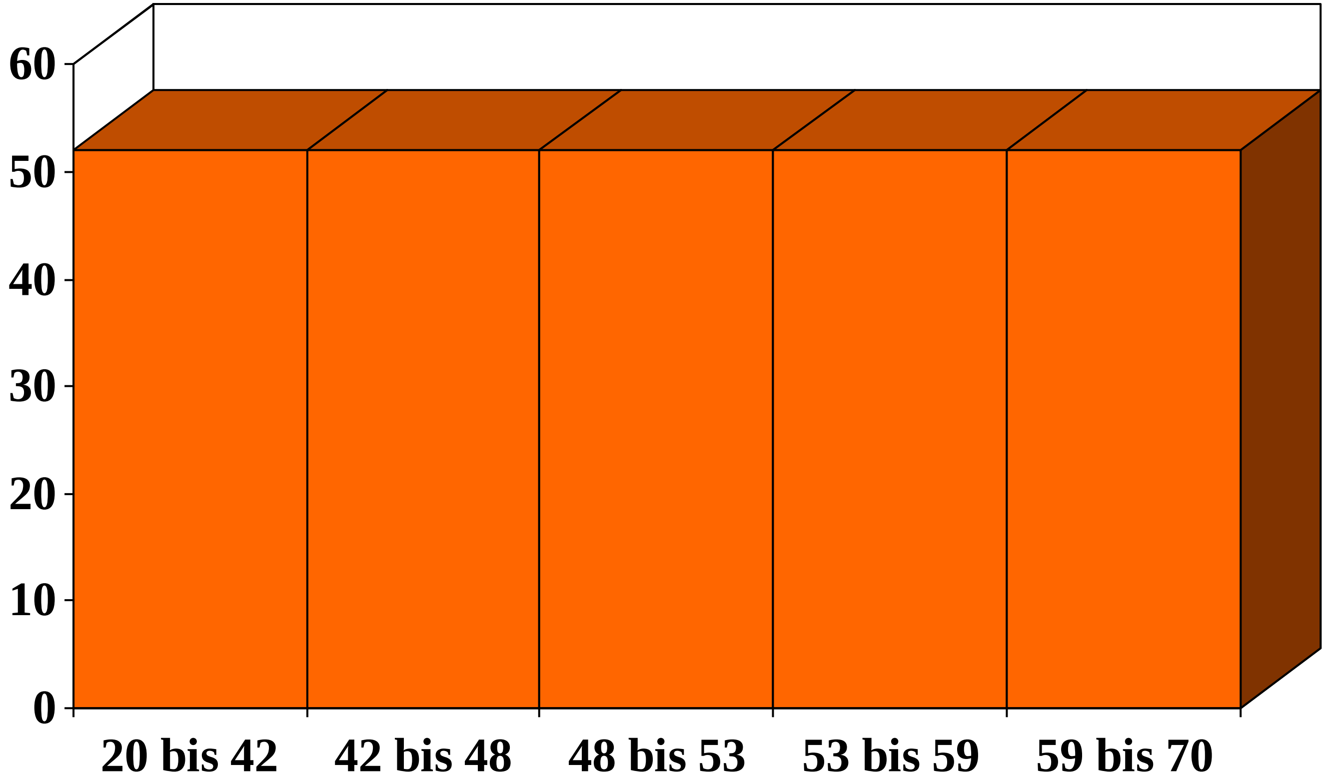
Equi-Width Histogram

SELECT * FROM person WHERE 25 < age < 40;



Equi-Depth Histogram

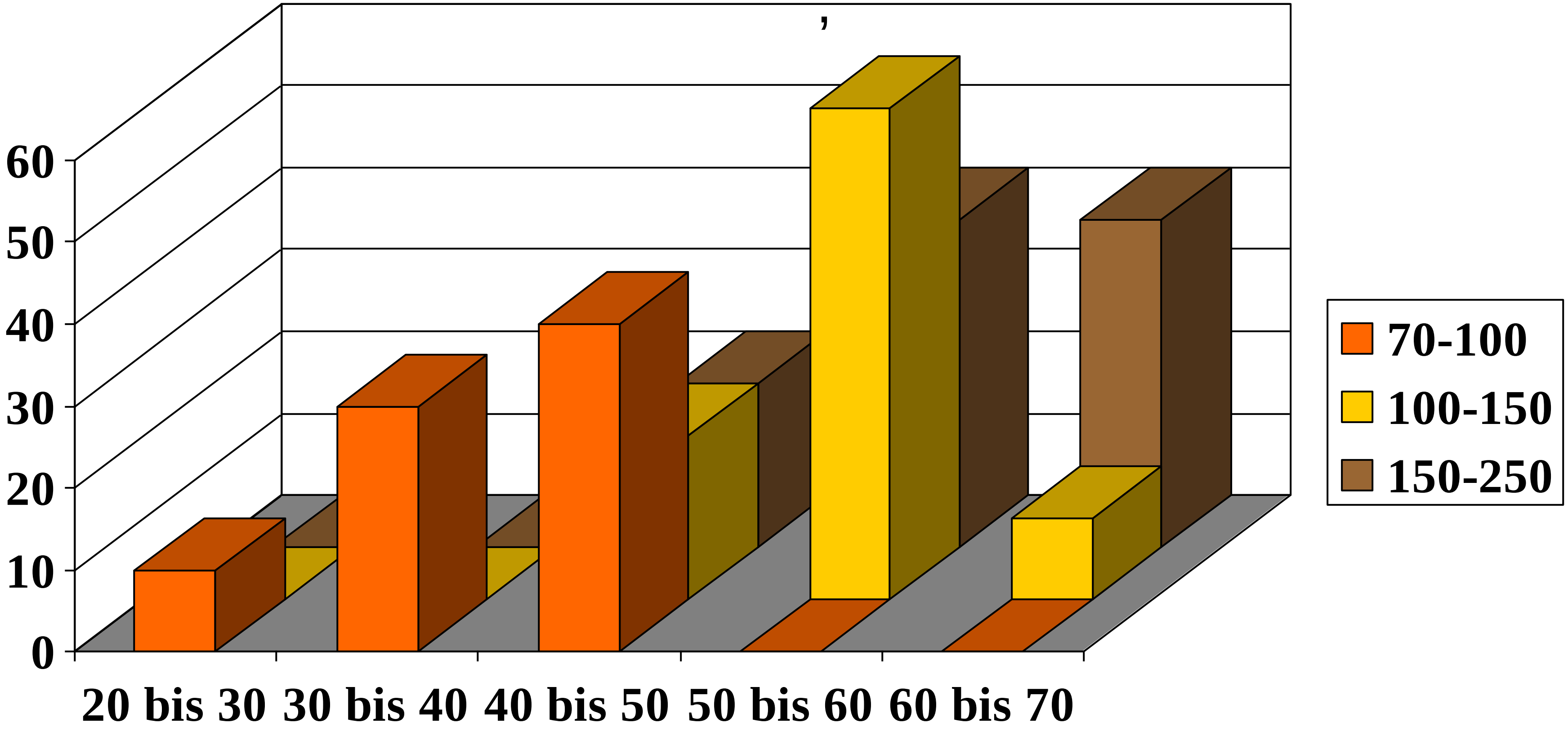
SELECT * FROM person WHERE 25 < age < 40;



Multi-Dimensional Histogram

SELECT * FROM person
WHERE 25 < age < 40 AND salary > 200;

;



Enumeration Algorithms

- Query Optimization is NP hard
  - even ordering or Cartesian products is NP hard
  - in general impossible to predict complexity for given query
- Overview of Algorithms
  - Dynamic Programming (good plans, exp. complexity)
  - Greedy heuristics (e.g., highest selectivity join first)
  - Randomized Algorithms (iterative improvement, Sim. An., …)
  - Other heuristics (e.g., rely on hints by programmer)
  - Smaller search space (e.g., deep plans, limited group-bys)
- Products
  - Dynamic Programming used by many systems
  - Some systems also use greedy heuristics in addition

Dynamic Programming

```
Function: find_join_tree_dp (q(R_1, ..., R_n))
for i = 1 to n do
    optPlan({R_i}) ← access_plans (R_i) ;
    prune_plans (optPlan({R_i})) ;
for i = 2 to n do
    foreach S ⊆ {R_1, ..., R_n} such that |S| = i do
        optPlan(S) ← ∅ ;
        foreach O ⊂ S do
            optPlan(S) ← optPlan(S) ∪
                possible_joins (optPlan(O), optPlan(S \ O));
        prune_plans (optPlan(S)) ;
return optPlan({R_1, ..., R_n}) ;
```

- access_plans: enumerate all ways to scan a table (indexes, …)
- join_plans: enumerate all ways to join 2 tables (algos, commut.)
- prune_plans: discard sub-plans that are inferior (cost & order)

Access Plans

- SELECT * FROM R, S, T WHERE R.a = S.a AND R.b = T.b ORDER BY R.c;
- Assume Indexes on R.a, R.b, R.c, R.d
  - scan(R): cost = 100; order = none
  - idx(R.a): cost = 100; order = R.a
  - idx(R.b): cost = 1000; order = R.b
  - idx(R.c): cost = 1000; order = R.c
  - idx(R.d): cost = 1000; order = none
- Keep blue plans only. Why?
  - And how can all that be? (Whole lecture on all this.)

Access Plans for S

- SELECT * FROM R, S, T WHERE R.a = S.a AND R.b = T.b ORDER BY R.c;

- Assume Indexes on S.b, S.c, S.d

  - scan(S): cost = 1000; order = none
  - idx(S.b): cost = 10000; order = none
  - idx(S.c): cost = 10000; order = none
  - idx(S.d): cost = 10000; order = none

Access Plans for T

- SELECT * FROM R, S, T WHERE R.a = S.a AND R.b = T.b ORDER BY R.c;

- Assume Indexes on T.a, T.b

  - scan(T): cost = 10; order = none
  - idx(T.a): cost = 100; order = none
  - idx(T.b): cost = 100; order = T.b

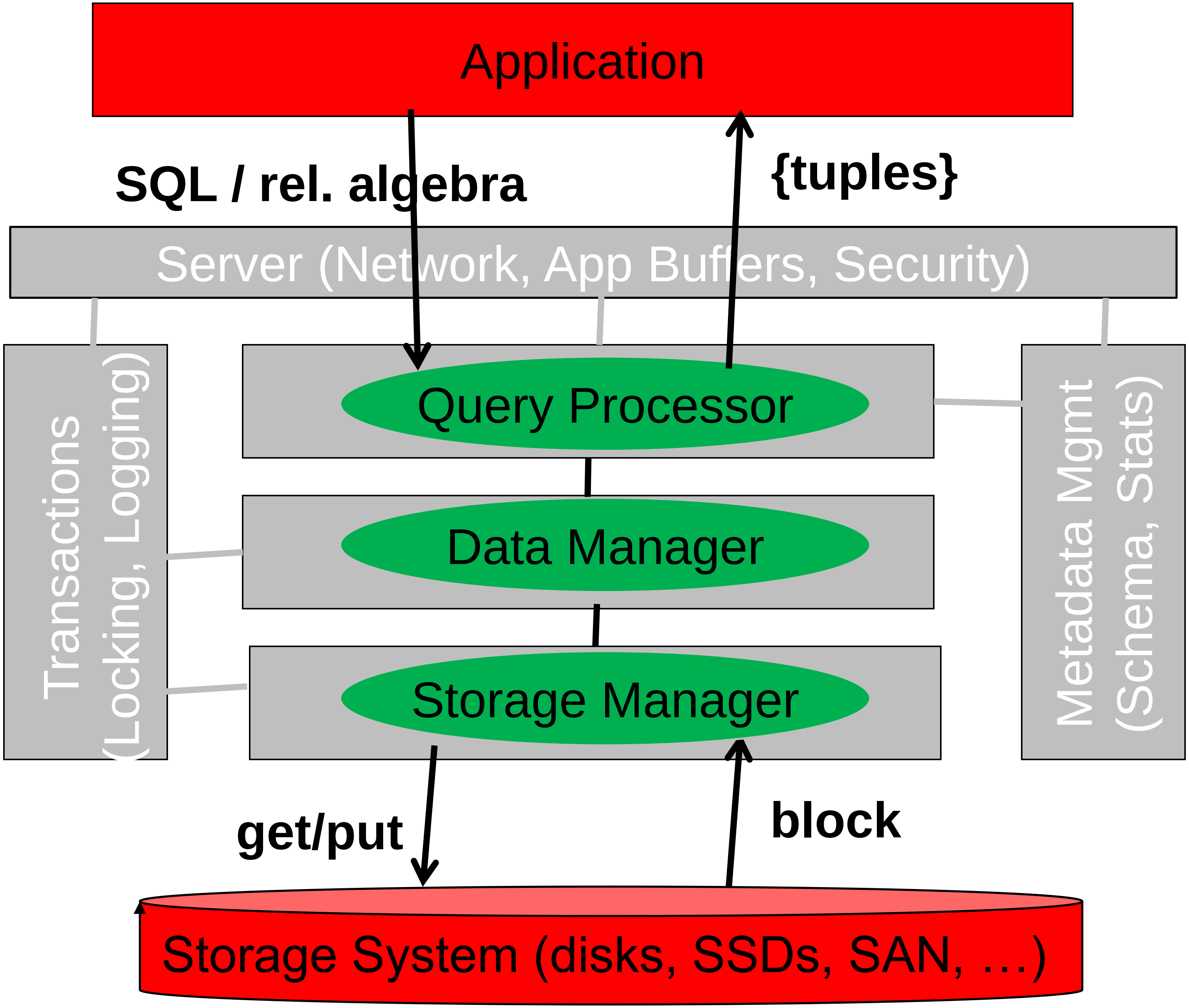
Join Plans for R join S

- SELECT * FROM R, S, T WHERE R.a = S.a AND R.b = T.b ORDER BY R.c;

- Consider all combinations of (blue) access plans
- Consider all join algorithms (NL, IdxNL, SMJ, GHJ, …)
- Consider all orders: R x S, S x R
- Prune based on cost estimates, interesting orders
- Some examples:
  - scan(R) NLJ scan(S): cost = 100; order = none
  - scan(S) IdxNL Idx(R.a): cost = 1000; order = none
  - idx(R.b) GHJ scan(S): cost = 150; order = R.b
  - idx(R.b) NLJ scan(S): cost = 250; order = R.b

Join Plans for three-way (+) joins

- SELECT * FROM R, S, T WHERE R.a = S.a AND R.b = T.b ORDER BY R.c;
- Consider all combinations of joins (assoc., commut.)
  - e.g., (R A S) A T, S A (T A R), ...
  - sometimes even enumerate Cartesian products
- Use (pruned) plans of prev. steps as building blocks
  - consider all combinations
- Prune based on cost estimates, interesting orders
  - interesting orders for the special optimality principle here
  - gets more complicated in distributed systems
- Exercise: Space and Time complexity of DP for DBMS.

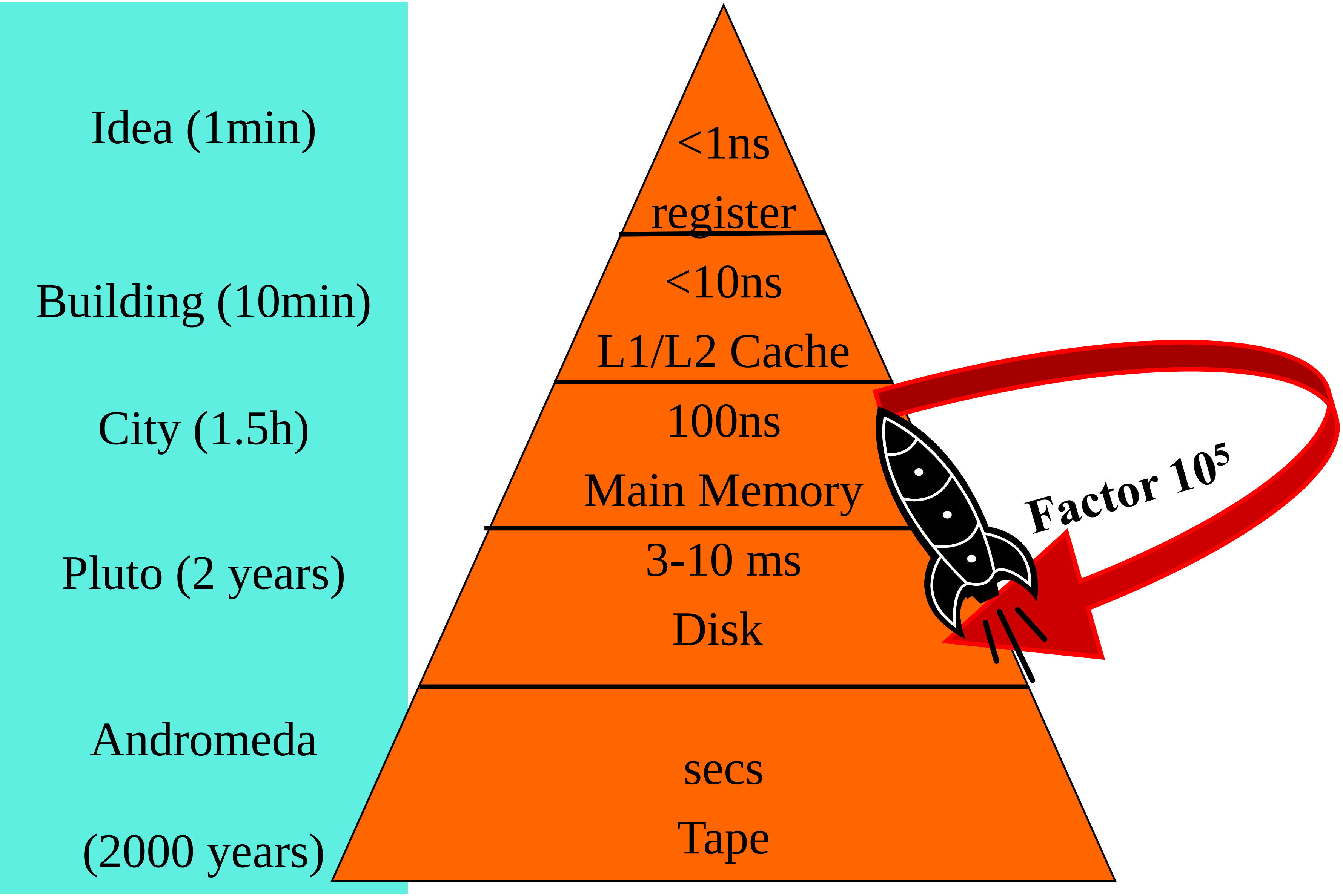


Application
SQL / rel. algebra
{tuples}
Server (Network, App Buffers, Security)
Transactions
(Locking, Logging)
Query Processor
Data Manager
Storage Manager
Metadata Mgmt
(Schema, Stats)
get/put
block
Storage System (disks, SSDs, SAN, …)

Storage System Basics

- Storage is organized in a hierarchy
  - combine different media to mimic one ideal storage
- Storage systems are distributed
  - disks organized in arrays
  - cloud computing: DHT over 1000s of servers (e.g., S3)
  - advantages of distributed storage systems
    - cost: use cheap hardware
    - performance: parallel access and increased bandwidth
    - fault tolerance: replicate data on many machines
- Storage access is non uniform
  - multi-core machines with varying distance to banks
  - sequential vs. random on disk and SSDs
  - place hot data in the middle of disk

Storage Hierarchy



Why a Storage Hierarchy?

- Mimics ideal storage: *speed of register at cost of tape*
  - unlimited capacity // tape
  - zero cost // tape
  - Persistent // tape
  - zero latency for read + write // register
  - infinte bandwidth // register
- How does it work?
  - Higher layer „buffers" data of lower layer
  - Exploit spatial and temporal locality of applications

Disks: Sequential vs. Random IO

- Time to read **1000 blocks** of size **8 KB**?

- Random access:

$$t_{rnd} = 1000 * t$$
$$= 1000 * (t_s + t_r + t_{tr}) = 1000 * (10 + 4.17 + 0.16)$$
$$= 1000 * 14.33 = \mathbf{14330\ ms}$$

- Sequential access:

$$t_{seq} = t_s + t_r + 1000 * t_{tr} + N * t_{\text{track-to-track seek time}}$$
$$= t_s + t_r + 1000 * 0.16\ ms + (16 * 1000)/63 * 1\ ms$$
$$= 10\ ms + 4.17\ ms + 160\ ms + 254\ ms \approx \mathbf{428\ ms}$$

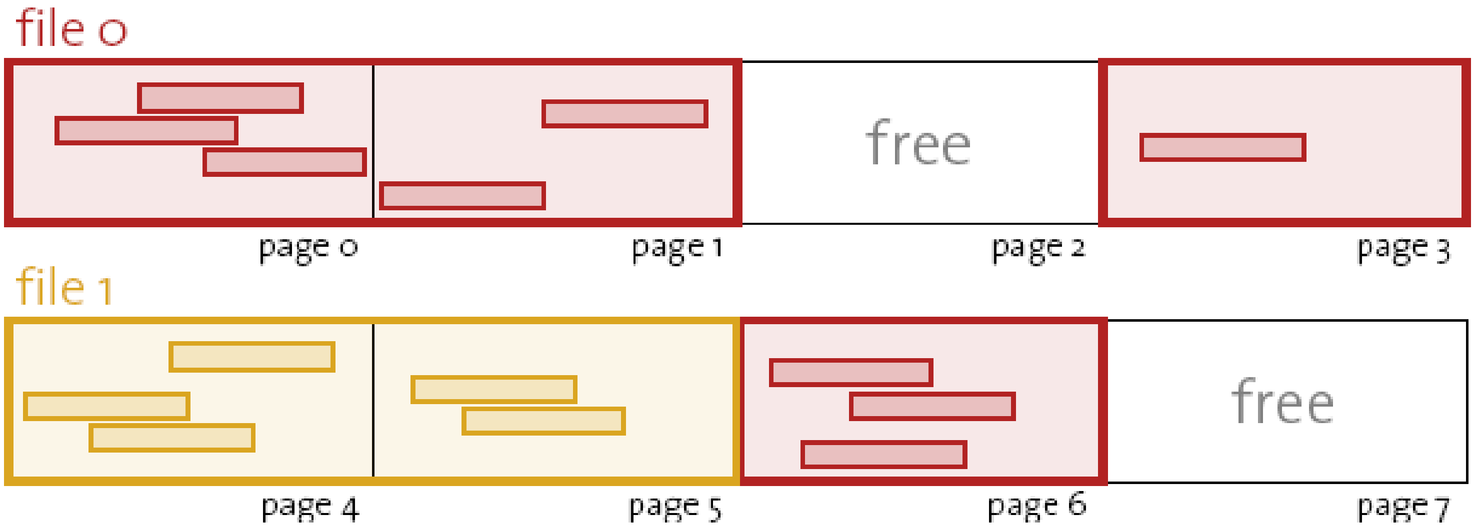
- **Need consider this gap in algorithms!**

[Information Systems Class]

Storage Manager

- Control all access to external storage (i.e., disks)
  - implements external storage hierarchy (SSD, tape, disks, …)
  - optimize heterogeneity of storage
  - outsmarts file system: operating system caching
  - write-ahead logging for redo and undo recovery
  - Oracle, Google, etc. implement their own file system
- Management of files and blocks
  - keep track of files associated to the database (catalog)
  - group set of blocks into pages (granularity of access)
- Buffer management
  - segmentation of buffer pool
  - clever replacement policy; e.g., MRU for sequential scans
  - pin pages (no replacement while in use)

Database = { files }



- A file = variable-sized sequence of blocks
  - Block is the unit of transfer to disk. Typically, 512B
- A page = fixed-sized sequence of blocks.
  - A page contains records or index entries
  - (special case blobs. One record spans multiple pages)
  - typical page size: 8KB for records; 16 KB for index entries
  - Page is logical unit of transfer and unit of buffering
    - Blocks of same page are prefetched, stored on same track on disk

Table Spaces, Files, and Tables

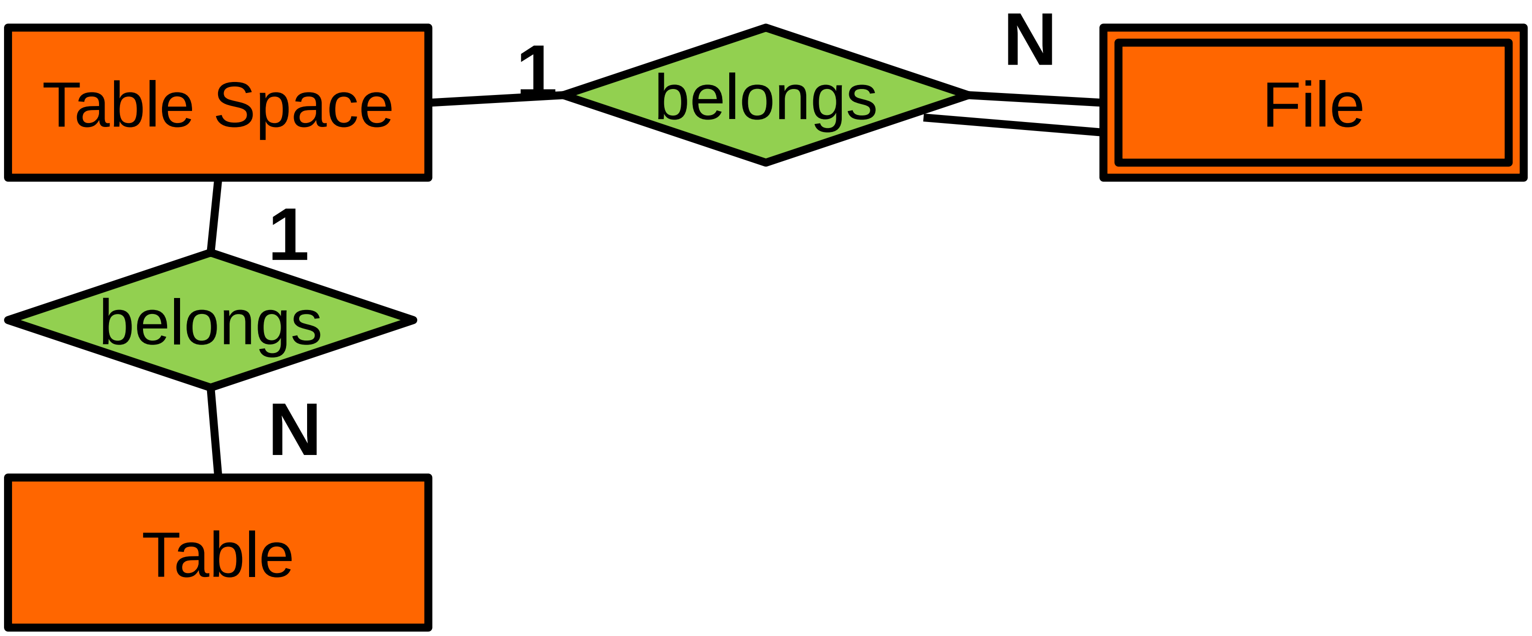


Table Spaces

- Explicit management of files allocated by the database
  - Each file belongs to a table space
  - Each table stored in a table space
  - Cluster multiple tables in the same file
- DDL Operations

```
CREATE TABLESPACE wutz
  DATAFILE a.dat SIZE 4MB, b.dat SIZE 3MB;
ALTER TABLESPACE wutz ADD DATAFILE ...;
CREATE TABLE husten TABLESPACE wutz;
```

- Warning: Full table space crash the DBMS
  - Classic emergency situation in practice

Buffer Management

- Keep pages in main memory as long as possible
  - Minimize disk I/Os in storage hierarchy

- Critical questions
  - Which pages to keep in memory (replace policy)?
  - When to write updated pages back to disk (trans. mgr.)?

- General-purpose replacement policies
  - LRU, Clock, … (see operating systems class)
  - LRU-k: replace page with *k* th least recent usage
  - 2Q: keep two queues: hot queue, cold queue
    - Access moves page to hot queue
    - Replacement from cold queue

Access Patterns of Databases

- Sequential: table scans

  $P_1, P_2, P_3, P_4, P_5, \ldots$

- Hiearchical: index navigation

  $P_1, P_4, P_{11}, P_1, P_4, P_{12}, P_1, P_3, P_8, P_1, P_2, P_7, P_1, P_3, P_9, \ldots$

- Random: index lookup

  $P_{13}, P_{27}, P_3, P_{43}, P_{15}, \ldots$

- Cyclic: nested-loops join

  $P_1, P_2, P_3, P_4, P_5, P_1, P_2, P_3, P_4, P_5, P_1, P_2, P_3, P_4, P_5, \ldots$

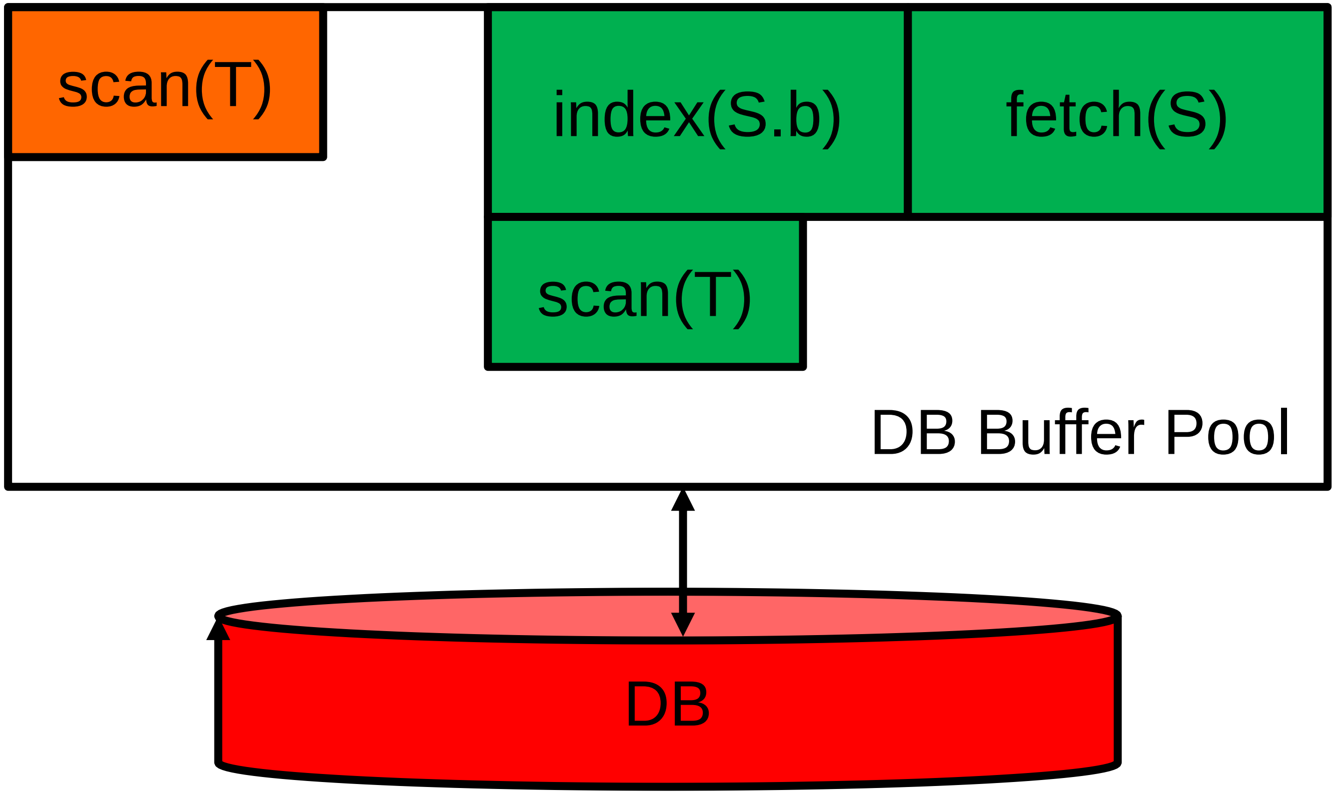
DBMin [Chou, DeWitt, VLDB 1985]

- Observations
  - There are many concurrent queries
  - Each query is composed of a set of operators
  - Allocate memory for each operator of each query
  - Adjust replacement policy according to access pattern
- Examples
  - scan(T): 4 pages, MRU replacement
  - indexScan(X)
    - 200 pages for Index X, LRU replacement
    - 100 pages for Table T, Random replacement
  - hashJoin(R, S)
    - 200 pages for Table S
- Allows to do load control and further optimizations
  - Economic model for buffer allocation, priority-based BM, …

DBMin: Buffer Segmentation

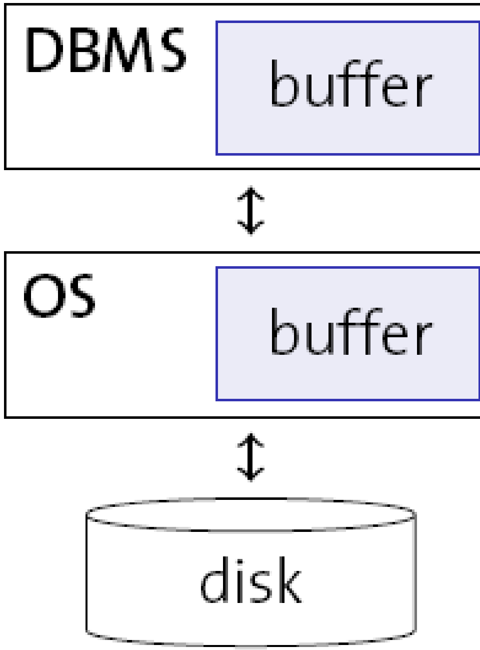
SELECT *
FROM T

SELECT *
FROM T, S
WHERE T.a=S.b



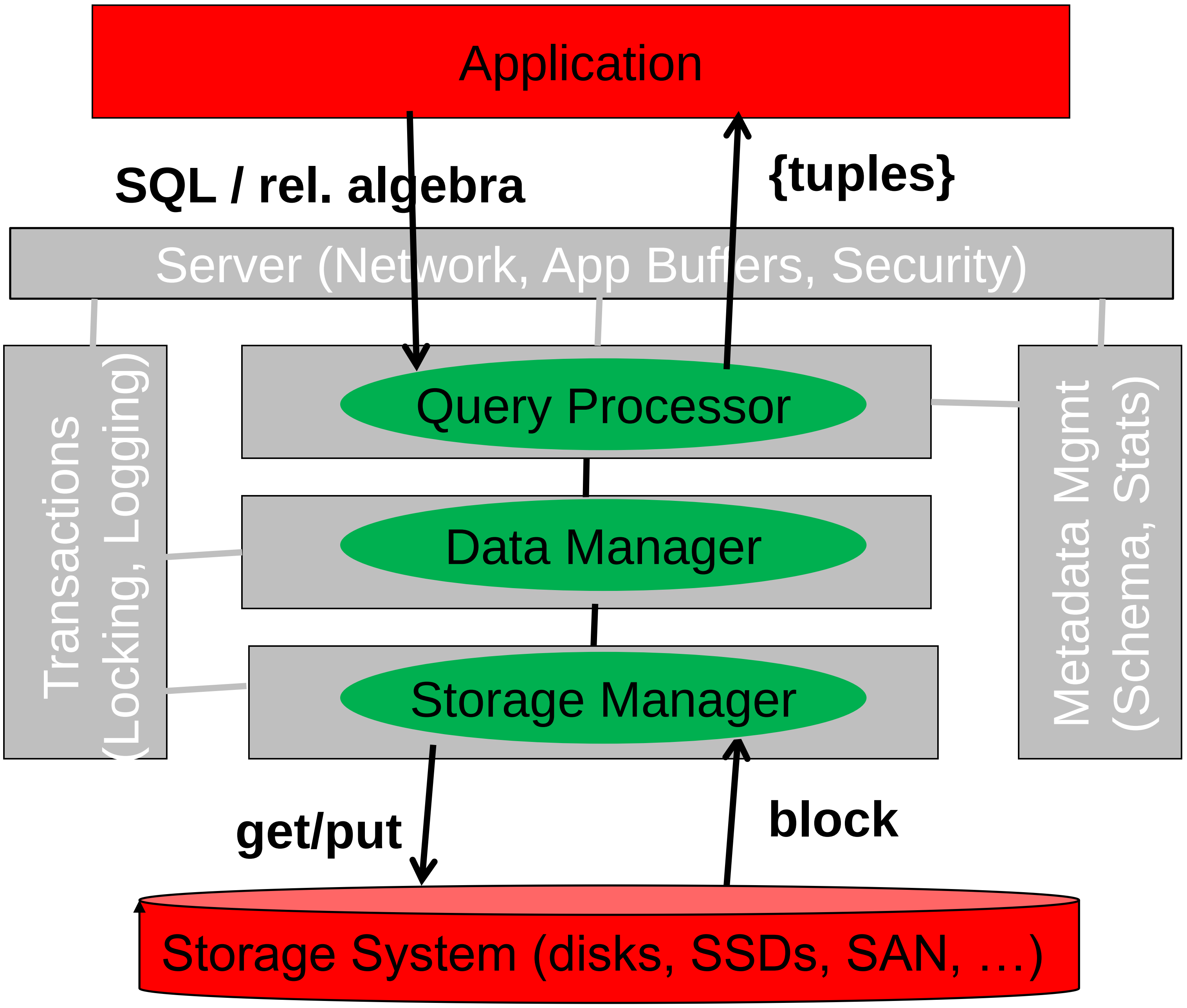
DBMS vs. OS: Double Page Fault

- DBMS needs Page X
  - Page X is not in the DB buffer pool
- DBMS evicts Page Y from DB buffer pool
  - make room for X
  - But, Page Y is not in the OS Cache
- OS reads Page Y from disk (swap)



- Summary
  - Latency: need to wait for (at least) two I/Os
  - Cost: If Y updated, up to three I/Os to write Y to disk
  - Utilization: Same page held twice in main memory

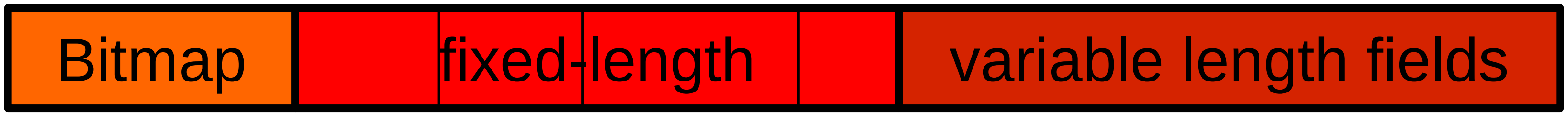
- If you are interested in DB/OS co-design, … ☺


Application
SQL / rel. algebra
{tuples}
Server (Network, App Buffers, Security)
Transactions
(Locking, Logging)
Query Processor
Data Manager
Storage Manager
Metadata Mgmt
(Schema, Stats)
get/put
block
Storage System (disks, SSDs, SAN, …)

Data Manager

- Maps records to pages
  - implement „record identifier" (RID)
- Implementation of Indexes
  - B+ trees, R trees, etc.
  - Index entry ~ Record (same mechanism)
- Freespace Management
  - Index-organized tables (IOTs)
  - Various schemes
- Implementation of BLOBs (large objects)
  - variants of position trees

Structure of a Record

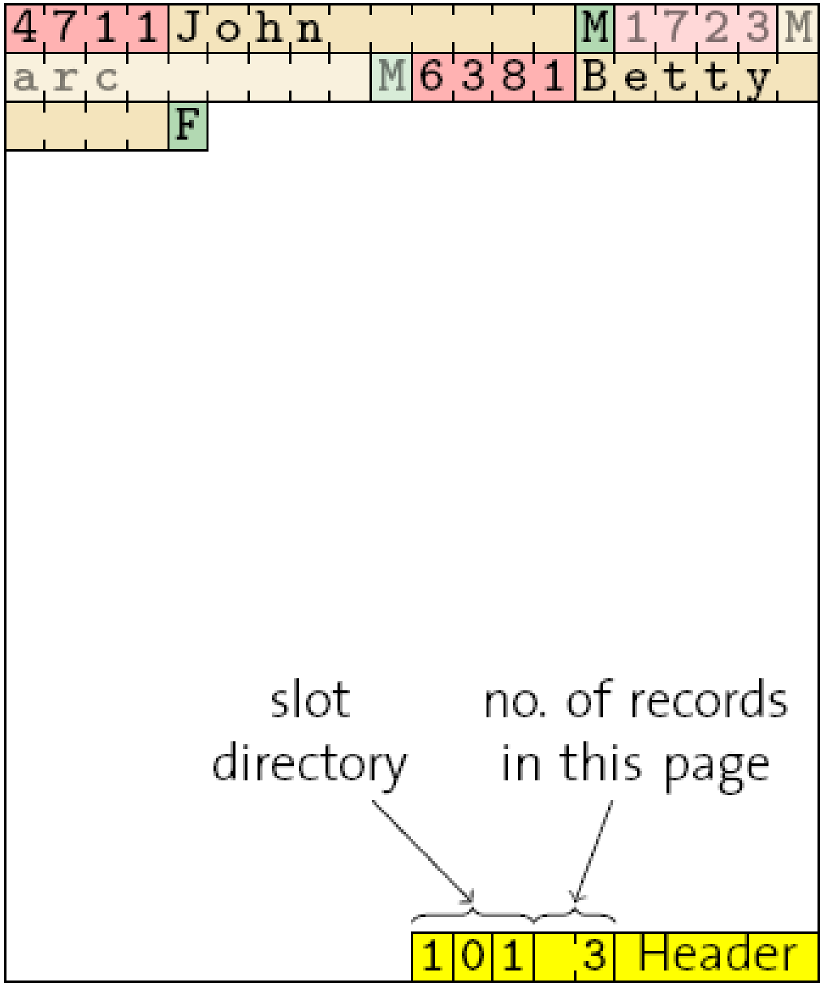


- Fixed length fields
  - e.g., number(10,2), date, char[100]
  - direct access to these fields
- Variable length fields
  - e.g., varchar[100]
  - store (*length, pointer)* as part of a fixed-length field
  - store payload information in a variable-length field
  - access in two steps: retrieve pointer + chase pointer
- NULL Values
  - Bitmap: set 1 if value of a field is NULL

Inside a Page

ID	NAME	SEX
4711	John	M
~~1723~~	~~Marc~~	~~M~~
6381	Betty	F

- **record identifier (rid):**
  <pageno, slotno>
  indexes use rids to ref. records
- record position (in page):
  slotno x bytes per slot
- records can move in page
  if records grow or shrink
  if records are deleted
  no need to update indexes



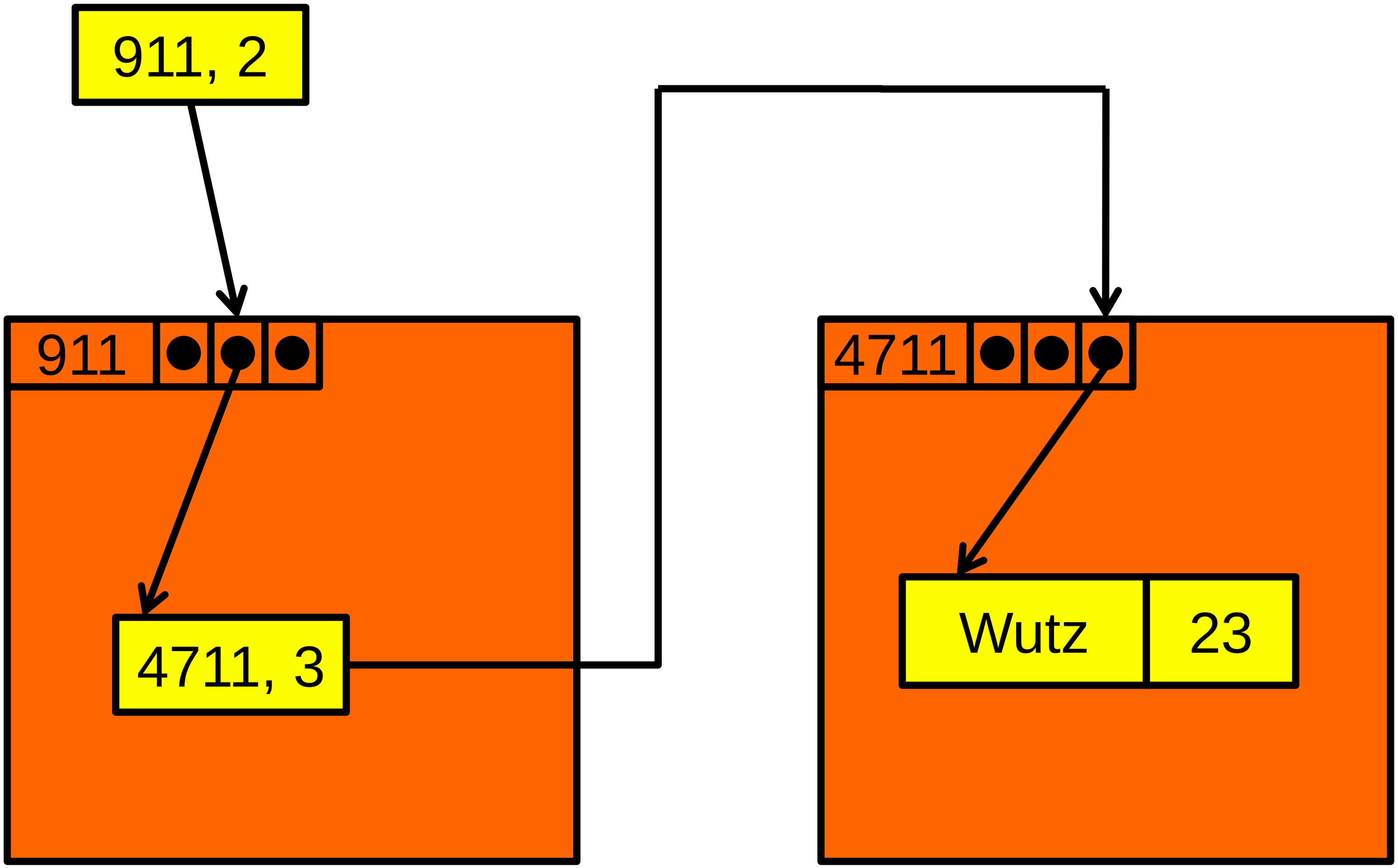
What happens when a page is full?

- Problem: A record grows because of an update
  - E.g., a varchar field is updated so that record grows

- Idea: Keep a placeholder (TID)
  - Move the record to a different page
  - Keep a „forward“ (TID) at „home“ page
  - If record moves again, update TID at „home“ page

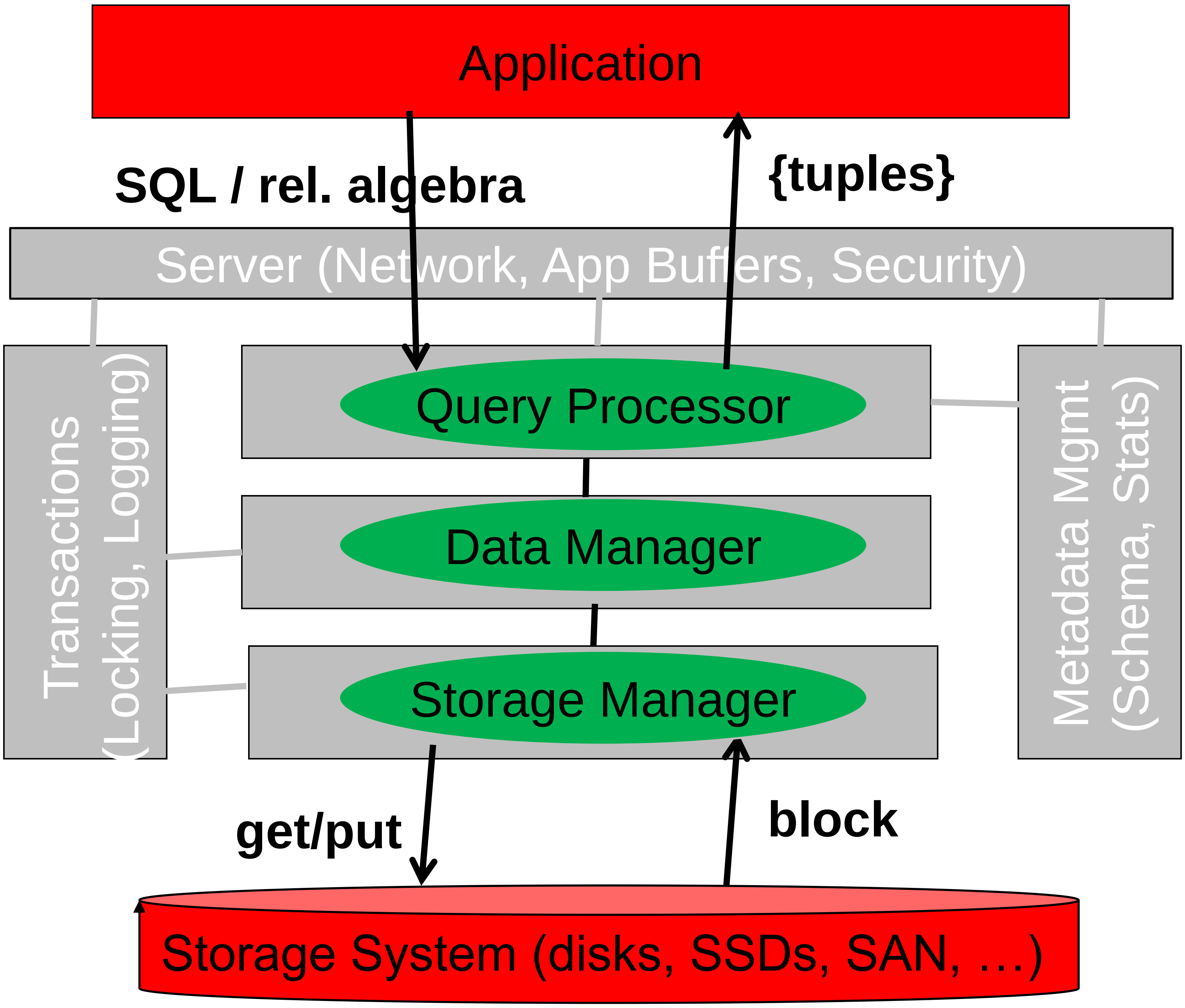
- Assessment
  - At most two I/Os to access a record; typically, only one
  - Flexibility to move records within and across pages
  - No need to update references to record (i.e., indexes)

TID (Forwarding) Concept



Freespace Management

- Find a page for a new record
  - Many different heuristics conceivable
  - All based on a list of pages with free space
- Append Only
  - Try to insert into the last page of free space list.
  - If no room in last page, create a new page.
- Best Fit
  - Scan through list and find min page that fits.
- First Fit, Next Fit
  - Scan through list and find first / next fit
- Witnesses: Classify buckets
- IOT: organize all tuples in a B+ tree
  - Let the B+ tree take care of splitting and freespace mgmt.


Application
SQL / rel. algebra
{tuples}
Server (Network, App Buffers, Security)
Transactions
(Locking, Logging)
Query Processor
Data Manager
Storage Manager
Metadata Mgmt
(Schema, Stats)
get/put
block
Storage System (disks, SSDs, SAN, …)

Meta-data Management: Catalog

- All meta-data stored in tables
  - Accessed internally using SQL
  - Eat your own dogfood
- Kinds of meta-data
  - Schema: used to compile queries
  - Table spaces: files used to store database
  - Histograms: estimate result sizes; query optimization
  - Parameters (cost of I/O, CPU speed, ...): query optimization
  - Compiled Queries: used for (JDBC) PreparedStatements
  - Configuration: AppHeap Size, Isolation Level, ...
  - Users (login, password): used for security
  - Workload Statistics: index advisors

What does a Database System do?

- Input: SQL statement
- Output: {tuples}

1. *Translate SQL into a set of get/put req. to backend storage*
2. *Extract, process, transform tuples from blocks*

- Tons of optimizations
  - Efficient algorithms for SQL operators (hashing, sorting)
  - Layout of data on backend storage (clustering, free space)
  - Ordering of operators (small intermediate results)
  - Semantic rewritings of queries
  - Buffer management and caching
  - Parallel execution and concurrency
  - Outsmart the OS
  - Partitioning and Replication in distributed system
  - Indexing and Materialization
  - Load and admission control
- + Security + Durability + Concurrency Control + Tools

Database Optimizations

- Query Processor (based on statistics)
  - Efficient algorithms for SQL operators (hashing, sorting)
  - Ordering of operators (small intermediate results)
  - Semantic rewritings of queries
  - Parallel execution and concurrency
- Storage Manager
  - Load and admission control
  - Layout of data on backend storage (clustering, free space)
  - Buffer management and caching
  - Outsmart the OS
- Transaction Manager
  - Load and admission control
- Tools (based on statistics)
  - Partitioning and Replication in distributed system
  - Indexing and Materialization

DBMS vs. OS Optimizations

- Many DBMS tasks are also carried out by OS
  - Load control
  - Buffer management
  - Access to external storage
  - Scheduling of processes
  - …
- What is the difference?
  - DBMS has intimate knowledge of workload
  - DBMS can predict and shape access pattern of a query
  - DBMS knows the mix of queries (all pre-compiled)
  - DBMS knows the contention between queries
  - OS does generic optimizations
- ***Problem: OS overrides DBMS optimizations!***

What to optimize?

Feature	Traditional	Cloud
Cost [$]	fixed	optimize
Performance [tps, secs]	optimize	fixed
Scale-out [#cores]	optimize	fixed
Predictability [σ($)]	-	fixed
Consistency [%]	fixed	???
Flexibility [#variants]	-	optimize

Put $ on the y-axis of your graphs!!!

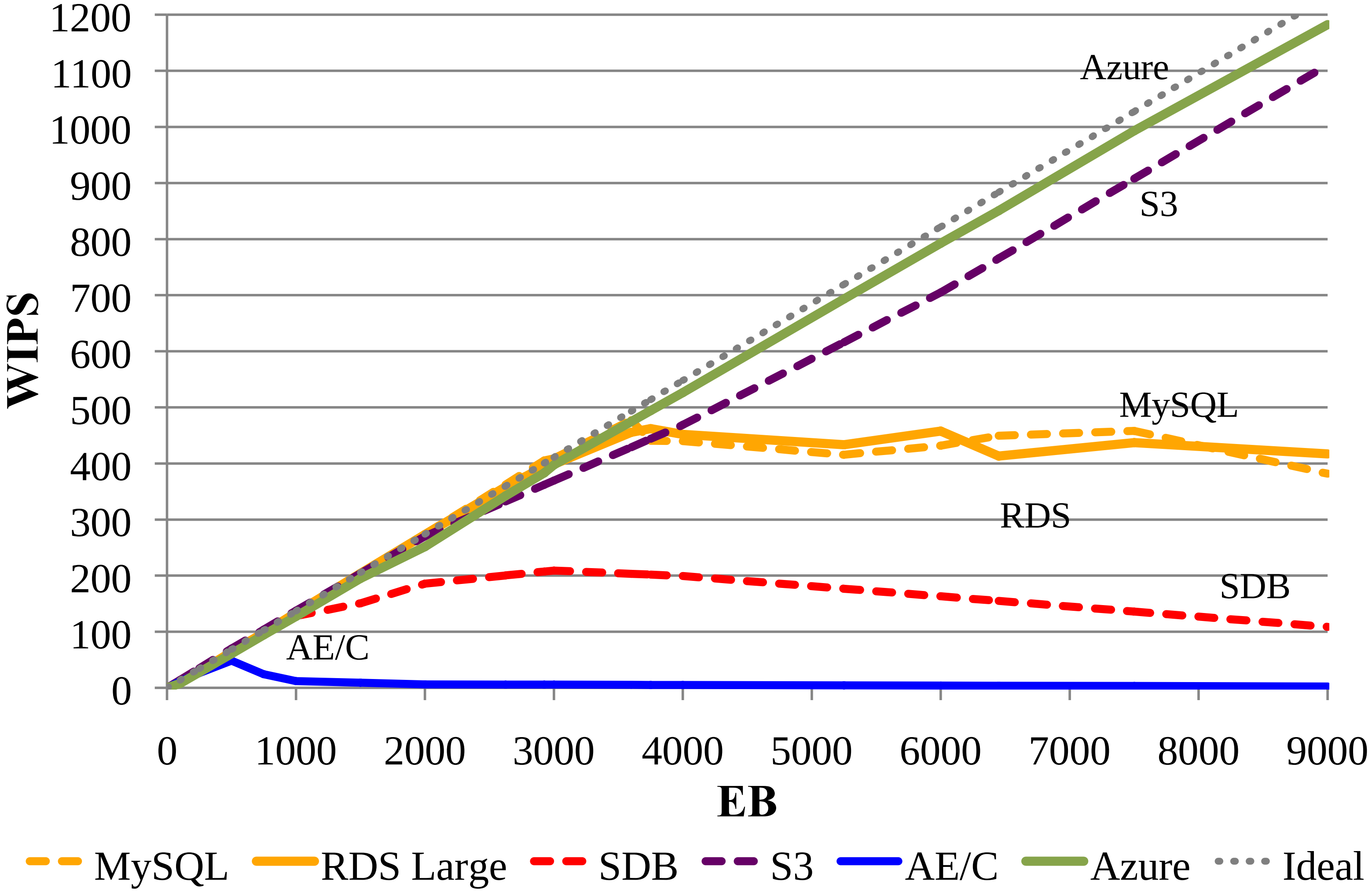
[Florescu & Kossmann, SIGMOD Record 2009]

Experiments *[Loesing et al. 2010]*

- TPC-W Benchmark
  - throuphput: WIPS
  - latency: fixed depending on request type
  - cost: cost / WIPS, total cost, predictability

- Players
  - Amazon RDS, SimpleDB
  - S3/28msec *[Brantner et al. 2008]*
  - Google AppEngine
  - Microsoft Azure

Scale-up Experiments



Cost / WIPS (m$)

	Low Load	Peak Load
Amazon RDS	1.212	0.005
S3 / 28msec	-	0.007
Google AE/C	0.002	0.028
MS Azure	0.775	0.005

What do you need for Project Part 2

- Storage Manager
  - Management of files
  - Simple buffer management
  - Free space management and new page allocation
- Data Manager
  - Slotted pages
- Query Processor
  - Implementation of scan, join, group-by
  - Iterator model
  - (external for extra credit)
- Catalog, Server, Transaction Manager
  - -
- **Be pragmatic! Get it running!**